1985

Threaded intermediate code /

Dale Edward Parson
Lehigh University

THREADED INTERMEDIATE CODE

by

Dale Edward Parson

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computing Science

Lehigh University

1985

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Oct 5, 1985
(date)

Samuel L. Gulde
Professor in Charge

Darrel J. Hillman
Head of Division

Eric D. O'Thomas
Chairman of Department

THREADED INTERMEDIATE CODE

TABLE OF CONTENTS

THREADED INTERMEDIATE CODE

Master of Science Thesis Abstract

by Dale E. Parson

Computing Science

An intermediate code is a binary representation of a program, including both instructions and data, which is produced by a translator from a human readable source program and which must be executed at run-time by way of some interpretation mechanism supported at the machine code level. A virtual computer is defined by the intermediate code interpreter; the instructions of the virtual computer correspond to the intermediate operation codes. Each intermediate operation code is supported by one or more sequences of machine code instructions which, when executed, yield results equivalent to the operation represented by the intermediate code element. Advantages of using intermediate codes include compactness of compiled operation code sequences, ease of compiler code generator design and increased compilation speed. The primary disadvantage is the increase in program execution time over pure machine code due to the operation of the run-time interpreter.

A powerful virtual machine can give a programming language designer an ideal target machine for the run-time support of the language being designed. The virtual

machine can be used by a computer designer to model and test the behavior of a hardware computer architecture being designed.

Some method for matching intermediate operation codes to supporting machine code instruction sequences must be used by the run-time interpreter. The varieties of threaded code utilize machine address pointers embedded in or referenced by the intermediate operation codes to direct interpretation to the correct machine code sequences. Types of threaded code discussed include subroutine-threaded code, direct-threaded code, indirect-threaded code and token-threaded code.

FORTH and LISP are supported by indirect-threaded code. Two FORTH language systems were installed with an editor, extended, tested and debugged as part of the research/design work of this thesis. In addition a subroutine library which supports data manipulation functions of LISP was designed, written in C language, tested and debugged. The package can be used by compiled C programs; it handles all pointer manipulation, error checking, storage allocation and garbage collection. Function names, actions and the syntax of data accepted by the package match those of full LISP.

THREADED INTERMEDIATE CODE

INTRODUCTION: INTERMEDIATE CODE AND VIRTUAL COMPUTERS

An intermediate code is defined to be a machine readable representation of a program, including both instructions and data, which is produced by a translator from a human readable source program and which must be executed at run-time by way of some interpretation mechanism supported at the machine code level.[1] Intermediate code is more compact and execution oriented than source code; it is a binary encoding used to reduce the size of representation and increase the speed of execution of the algorithm expressed in the source program beyond what would be possible with pure interpretation of program text. Intermediate code is not machine code, however, since it cannot be executed by the underlying hardware or microcode supported machine; this code serves as a data structure which guides the flow of machine code processing at run-time in a manner which is analogous to the way in which conventional machine code guides the flow of microcontrol execution in a microprogrammed

---

1. For a general discussion of intermediate codes and the virtual computers which they implement see Terrence W. Pratt, Programming Languages: Design and Implementation (Englewood Cliffs, N.J.: Prentice-Hall, 1984), p. 20-30.

machine at run-time.

It is instructive and relevant at this point to briefly review the mechanisms of translation and interpretation in order to develop a context for explaining how intermediate code meshes with these mechanisms and why it might be used.

Various schemes exist for taking computer programs written in a human readable source format and converting them into a representation suitable for machine execution. At one extreme lies pure translation, where source statements are fully expanded into equivalent machine code sequences, possibly with some optimization performed in order to increase execution speed or efficiency of memory use while preserving semantic intent.[2] On most modern computers the resulting target machine code is interpreted by a microprogram, the latter consisting of sequences of bit patterns in memory used to control the clocked gating of registers in the central processing unit onto circuit paths, through combinational circuitry, and thence into other registers.[3] A possible translation scheme is one where a source program is compiled into a microprogram

---

2. See Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design (Reading, Ma.: Addison-Wesley, 1977).

3. For a discussion of microprogrammed support of the conventional machine level see George D. Kraft and Wing N. Toy, Microprogrammed Control and Reliable Design of Small Computers (Englewood Cliffs, N.J.: Prentice-Hall, 1981).

target format without employing the conventional machine code level. Such an executable format would hold a run-time speed advantage over a machine code representation for the same source program, since regular machine code would be interpreted at run-time by general purpose microcode calling sequences; the decoding of machine code at run-time in order to determine which microcode segments are to be executed takes time. Several limitations on the use of microcode inhibit the general use of such a scheme. Writable control store (when it is available at all - control store is often of the read-only "firmware" variety) is usually composed of far fewer words than the main memory which holds conventional machine words. This word count difference allows control store to be designed using faster, hence more expensive memory circuits than that used for main store. Thus it is more economical to encode long gating sequences represented by large programs as machine code words in relatively slow main memory and interpret these encoded words by repeatedly using relatively short segments of expensive, fast memory bit patterns than to compile the source into microcode, a method which would require a large amount of expensive memory to hold the encoded program. Furthermore, the conventional machine level is normally designed to be more convenient to use than the microcode level for compiler writers. Put simply, compilers which produce machine code

are more easily designed than are compilers which produce microcode. Microcode reflects the structure of the machine's hardware; machine code at its best reflects the run-time structure of the programming language being translated by the compiler. The current trend in computer design is to implement higher-level machine code architectures than in the past in order to support procedural, block structured languages such as Pascal and C, as well as dedicated architectures to support run-time facilities for special-purpose languages such as LISP and various object oriented languages; often these architectures are implemented within a single integrated circuit processor package.[4] Thus the trend in hardware design is one of easing the job of the compiler designer by presenting him with a more reasonable machine code architecture with more efficient execution characteristics than previously available, obliterating any reason for him to want to produce microcode.

At the extreme opposite of pure translation lies pure interpretation.[5] An interpreter uses the source

---

4. For an up-to-date view of trends in the design of computer hardware and architectures see Computer Design, Vol. 23, No. 10 (September, 1984). The issue is entitled, "Future Computers: System Design Beyond 1984."

5. Interpretation is discussed by Andrew S. Tanenbaum in Structured Computer Organization (Englewood Cliffs, N.J.: Prentice-Hall, 1976), p. 344-346.

version of the program at run-time, "executing" it by examining the source code and using what it finds to steer execution through equivalent code sequences which are part of the interpreter's collection of code routines. A pure interpreter requires lexical and syntactic analysis at run-time with repeated analysis required for code sequences which are executed more than once. Most interpreters are not pure since they do some condensation and encoding of the source program, but many still use a format very close in organization to that of the source program; for many interpreters only the lexical portion of translation occurs before run-time. The major penalty for using interpretation is loss of speed, since time is divided between executing code sequences equivalent to the source program and executing code needed to interpret the source program; in a compiled program the source interpretation component does not exist at program run-time. Further speed penalties are experienced because optimizations which can be performed during translation cannot be employed by interpretation; the code routines of the interpreter are created when the interpreter is created and are of general purpose nature, hence are not optimized for the specific user source program; code generated by an optimizing compiler is tuned to the user's source program to a degree. Also all of the code library routines of an interpreter must be available, hence loaded

into memory when an interpreter is used, even though some of these may never be called during the process of interpreting a specific program. A major advantage of working with an interpreter is its ease of use - the run-time presence of source or near-source code allows for program-entry-time and run-time error messages which supply information about errors in terms of the source program. Interpreters are generally interactive and often more user-friendly than translators.

> "The ultimate in internal program conciseness is not, however, the machine code right at the end of the slide. It is, instead, an ideal machine code for running the source language. This ideal machine code has just the operations the source language needs, and these are encoded in the most consise way - the more frequently used needing fewer bits than the less frequently used." [6]

Intermediate code occupies the middle ground between totally translated machine code and totally interpreted source code; it exhibits some of the characteristics of both of these. To the compiler writer intermediate code appears to be a powerful target machine code with potent, high-level instructions which readily meet the run-time requirements of the language being compiled; the apparent underlying computer whose instruction set consists of the operations of the intermediate code is called the

---

6. P. J. Brown, Writing Interactive Compilers and Interpreters (New York, N.Y.: John Wiley and Sons, 1979), p. 51.

virtual computer supported by the intermediate code.

> "A hardware computer is termed an actual computer. A computer that is partially or wholly simulated by software or microprograms is termed a virtual computer. When a programming language is implemented, the run-time data structures and algorithms used in program execution define a computer. Because this computer is almost always at least partially software-simulated, we speak of this as the virtual computer defined by the language implementation. The machine language of this virtual computer is the executable program form produced by the translator for the language, which may take the form of actual machine code if the language is compiled, or, alternatively, may have some arbitrary structure if the language is interpreted." [7]

As an example, a stack machine may be implemented as a virtual computer. The instruction set for this computer would include arithmetic instructions which would pop operands off of a stack, perform arithmetic operations using these values, and push results back onto a stack. A virtual computer could contain instructions for performing such operations as taking the square root or common logarithm of a floating point number, even though the machine code level of the computer upon which the virtual computer is built does not have any such operations in its instruction set. All that is required is that some sequence of machine code instructions when executed will yield a result equivalent to what the virtual operation would have yielded.

---

7. Terrence W. Pratt, op. cit., p. 25-26.

Machine code sequences used to support virtual operations are known as "primitives"; they are the primitive operations of the underlying machine. Intermediate code sequences are sometimes known as "secondaries." At run-time an interpreter for the virtual computer uses the virtual instructions to select what sequences of primitives to execute.[8] Alternatively, some virtual instructions may require execution of some other, usually simpler sequences of virtual instruction in order to acheive instruction simulation; in such a case, the interpretation process must be applied recursively to these intermediate levels of intermediate code. Eventually, however, the interpretation process must lead to the eventual execution of equivalent machine code sequences. It is possible to support multiple levels of different virtual computers with different instruction sets. Each intermediate code would appear as machine code to the level above it; calls to interpreters would be nested with the next lower level interpreter called for the interpretation of the next lower level virtual computer, until again the actual machine code level is reached. This discussion will be limited to schemes with a single level of intermediate code between the source code and the

---

8. See R.G.Loeliger, Threaded Interpretive Languages (Peterborough, N.H.: BYTE Publications, 1981), p. 7 for an introductory discussion of primitives and secondaries.

machine code levels; the principles remain the same for multi-level code indirection.

Much of the preceding elaboration was performed with the intention of establishing a context within which to discuss the advantages and disadvantages of using intermediate code schemes over strict translation or interpretation in the processing of computer programs.

A prime advantage of using an intermediate code virtual computer is that it reduces the complexity of the compiler for a given language. Since the architecture of the virtual computer is designed to support the run-time requirements of the programming language, compilation is simpler. Less code generation need be done, making the compilation process faster and less error prone. The intermediate code represents an optimized instruction set for the language, so optimization of generated code is more easily acheived.

Portability of the compiled code is a related issue.[9] If a compiler produces intermediate code and that intermediate code can be interpreted on a variety of computing machines, then programs distributed as translated intermediate code segments can be executed on all of those computing machines; the user must, of course, have the

---

9. See Andrew S. Tanenbaum, op. cit., p. 354-364 for a discussion of program portability across multiple virtual machines.

intermediate code interpreter for his underlying machine.

It is possible to write compilers which will translate different source languages into the same intermediate code, making the intermediate code level a level of junction between differing computers and differing languages. When a compiler for a new language is written to produce intermediate code, then that language will become available on a variety of machines. It is important, however, not to try to support too varied a collection of programming languages with a single virtual computer.

> "An intermediate language that must support several languages must contain the union set of all those language features, which will make the intermediate language considerably more complicated than it would be for any one of the languages alone. Many of the features must be identified as peculiar to one or another language because of semantic differences. Some language features, particularly the declaration, I/O, and block structuring conventions, require such specialized treatment that there might as well be a different intermediate language for each language.
>
> "Some problems arise in the design of back ends with multiple machines and one intermediate language, but they are less severe than the multiple language problem. If the intermediate language is consistently designed, each of its operations can somehow be implemented on a target machine of reasonable capability." [10]

For the programming student use of an intermediate language scheme is advantageous since the intermediate

---

10. William A. Barrett and John D. Couch, Compiler Construction: Theory and Practice (Chicago, IL.: Science Research Associates, 1979), p. 472.

mechanisms often retain references to source program information such as data types and line numbers of instructions which can be useful in helping to find the location of bugs in a program. An intermediate code level supports the creation of an interactive program execution environment.[11] As already stated, the compilation process is faster than with a machine code compiler; student programs often undergo frequent compilation but infrequent execution after programming assignments are completed. Thus an intermediate code virtual machine offers students most of the advantages of pure interpretation without the severe run-time speed penalties suffered using the latter technique.

For the microprocessor controlled machine designer and computer hobbyist an important aspect of the use of intermediate codes is the reduction in memory requirements over what is possible with pure translation or interpretation. The reduction in memory requirements over interpretation is possible because the encoding technique is more compact than the use of verbose source text. The reduction in memory requirements over machine code translation is possible because each intermediate code instruction represents a large number of machine code primitive

---

11. See P. J. Brown, op. cit., for information concerning the advantages of creating an interactive run-time environment.

instructions. If the same intermediate code op codes are used repeatedly, then significantly less memory is used for their storage than would be required for the repeated generation of similar machine code sequences by a machine code producing translator. In virtual computers where all of the primitive code sequences are loaded at run-time, the user must pay the initial memory overhead of storing these primitives, many of which may never be used if the user program is short. Long user programs which do make use of the full capabilities of the virtual machine, on the other hand, would experience reductions in memory consumption over machine code schemes because of the compact encoding of powerful operations. Here we see the similarity between virtual computers supported by interpreters and machine code supported by microprogramming mentioned earlier; the technique used for encoding a powerful instruction architecture reduces the memory requirements for large compiled programs running on that architecture. FORTH, a threaded compiled language which will be discussed later in this essay, produces extremely memory efficient intermediate code.[12] I have seen it used on several different types of computer controlled equipment used in semiconductor manufacture for

---

12. See E. Rather, L. Brodie, and C. Rosenberg, Using FORTH (Hermosa Beach, CA.: FORTH, Inc., 1979).

this reason.

> "If programming languages were originally defined in terms of their virtual computers, so that each language was associated with a single commonly understood virtual computer, then description of the semantics of each language in terms of its virtual computer would be straightforward. Unfortunately, because languages are usually defined by giving a semantics for each syntactic construct individually, language definitions specify only implicitly an underlying virtual computer."[13]

Thus the virtual computer represented by the intermediate code instruction set provides a specification and implementation tool which can be used by the programming language designer and implementer in establishing semantics of a language and verifying semantic correctness of its implementations.

The final and in some ways most important use of virtual computers is in the area of experimentation with novel computer architectures. Here is where the similarity of machine code interpretation of intermediate code to microcode interpretation of machine code becomes significant: if the virtual computer design proves to be useful, design a microcoded computer to interpret the virtual machine's code as actual machine code at run-time. The LISP Machine[14] is just one example of a computer

---

13. Terrence W. Pratt, op. cit., p. 27.

14. See Bawden, Greenblatt, Holloway, Knight, Moon and Weinreb, "The LISP Machine," Artificial Intelligence: An MIT Perspective (Cambridge, MA.: MIT Press, 1979), p. 345-373.

originally implemented as an intermediate software construct but now available as an actual microcode supported machine. Using intermediate code to experiment with new architectures allows the designer to create, test, and debug techniques before going through the time and expense of committing them to hardware and firmware. When the archtectural design is satisfyingly complete then the hardware/microcode implementation can be begun. The IS-25 Assembly Language specification has been used to design a series of computers whose run-time characteristics support the efficient execution of compiled C programs,[15] but not before extensive experience with compiled C programs was gained using more intermediate level software support.

Shifting the interpretation of the virtual computer operations down into microcode removes the major liability experienced with the use of intermediate code, that of poor execution speed.

> "The interpretation procedure for execution of this translated program form must be represented by software because the hardware interpreter cannot be used directly. Use of a software interpreter ordinarily results in relatively slow program execution. In addition, languages which are software-interpreted also tend to require extensive software simulation of primitive operations, storage management, and other language

---

15. See 3B20S/A Computers - IS25 Assembler Manual (Winston-Salem, N.C.: AT&T Technologies, 1984).

> features. Translators for interpreted languages tend to be rather simple, with most of the complexity of the implementation coming in the simulation software." [16]

Execution time of an intermediate code program may be from two to many more times longer than the execution time of an algorithmically equivalent machine code program. The amount of time increase depends upon the complexity of the intermediate code virtual machine, the resultant complexity of the interpreter needed to support it, and the degree to which the architecture of the underlying machine code level meets the support requirements of the virtual machine architecture.

We can conclude from the above that the greatest utility for the use of intermediate code virtual computers is present in three situations. One situation is in an educational environment where frequent compilations take place but long term recurring execution of large compiled programs normally does not occur; the speed of compilation and interaction supporting nature of the virtual computer code is useful here. A second environment is one where experimentation with programming language design occurs; novel compilers can be designed to be relatively simple and reliable if they must compile code only to the powerful virtual machine level; such design can be accomplished in a fairly short amount of time. A third environment is one where experimentation with

16. Terrence W. Pratt, op. cit., p. 25.

machine architecture design occurs; the novel architecture can be simulated as a virtual computer before committing it to hardware and microcode. Thus the intermediate code virtual computer has its main utility as a software laboratory aid.

While most of the discussion so far has dealt with executable code, it is important to note that the virtual computer can support complex mechanisms for the structuring and translation of a program's data from a high level notation to an intermediate level storage scheme. More will be said on this in later sections.

In addition to study of the relevant literature, two research/design projects contributed to this thesis. One was the installation, modification, extension and debugging of two FORTH language systems, one on an Apple II Plus microcomputer[17] and another on the PDP-11/70 minicomputer at Albright College, Reading, Pennsylvania.[18] The second project was the design and implementation of a LISP data manipulation package on an AT&T 3B20S superminicomputer running UNIX System V at AT&T Technology Systems in Reading, Pennsylvania. The package presents C language

---

17. See W.F. Ragsdale, FIG-FORTH 6502 Assembly Source Listing (San Carlos, CA.: FORTH Interest Group, 1980).

18. See John S. James, FIG-FORTH for PDP-11 Assembly Source Listing (San Carlos, CA.: FORTH Interest Group, 1979).

programs with the capability of calling LISP data manipulation functions for performing symbolic processing. The FORTH and LISP programs will not be covered in exhaustive detail in this thesis; instead I will elaborate upon aspects of these systems relevant to the topic of threaded intermediate run-time code.

THREADED INTERMEDIATE CODE MECHANISMS AND STRUCTURES

The "threads" of threaded intermediate code are pointers. Threaded code forms a data structure with objects connected via pointers. Objects in the data structure may be composed of pointers to other such objects, pointers to machine code primitive routines, pointers to machine code level data items, pointers to more complex data structures, along with locally stored items of data. The essential features of a threaded intermediate code are that it consists of linked objects, and that a run-time interpreter executes portions of this code by following code links and initiating machine execution of the primitive routines to which these code links eventually lead.

> "All varieties of threaded code consist of a data structure that is a sequence of unique subroutine identifiers. Traditionally, threaded code has been kept close to the machine level and has included actual pointers to the subroutines (which themselves may be either intermediate language or machine code). Also traditionally,

> a portion of the processor resources - in particular, processor registers - has been dedicated to the use of the threaded code interpreter. As we shall see, neither absolute pointers nor register resources need be used to implement threaded code."[19]

The advantage to using such a type of intermediate encoding technique over other possible schemes is that the direct linkage of the threaded code to the equivalent machine code primitives via pointers allows for a very efficient run-time interpretation mechanism. Before categorizing the several varieties of threaded code and discussing their differences, let us examine a generic threaded code interpreter and analyze the workings of its component parts.[20]

A threaded code interpreter can be implemented using three procedures, procedures which we will call NEXT, CALL, and RETURN. In this discussion the program counter of the underlying machine code level machine will be called the PC; PC is the register used to fetch primitive level instructions. IP is the "instruction pointer" for the threaded code virtual machine; like PC, IP is used to fetch instructions, in its case intermediate level instructions. IP may consist of a processor register if

---

19. Terry Ritter and Gregory Walker, "Varieties of Threaded Code for Language Implementation," BYTE, Vol. 5, No. 9 (September, 1980), p. 211.

20. For a discussion of run-time interpretation of threaded code see R.G.Loeliger, op. cit., p. 18-25.

one is available; otherwise IP may be stored in memory and put into a register when necessary. Finally we must have access to a stack for storing return addresses during nested subroutine calls; if the machine level uses a stack for such purposes then that stack may be used, since all calls to machine primitive routines are nested within threaded code interpretation sequences; the only machine code to execute besides the operation supporting primitives is the interpreter mechanism itself.

Code sequence **NEXT** in the interpreter fetches the threaded code instruction indicated by IP into some temporary location in preparation for execution, after which **NEXT** advances IP to point to the next threaded code instruction; **NEXT** mirrors the way in which the micro-program uses the program counter at a lower level. If the threaded code sequence consists of a contiguous array of objects which themselves are composed of strictly encoded instructions, then the operations of **NEXT** are simply: 1) fetch the code word indicated by the IP, and 2) increment IP to point to the next operation code. This is normally the way the PC is advanced at the machine level. If, on the other hand, the item at which the IP points is not a simple intermediate operation code, but rather a more complex intermediate data structure, then the fetch portion of **NEXT**'s operation becomes more involved; **NEXT** must extract the instruction code from its place in

the data structure. Furthermore, if the instruction code is part of a complex data structure and these structures are stored in a linear array format then NEXT must advance IP to point to the next complete structure. Alternatively, the structures may not be stored in an array format but rather in a linked-list format with one of the structure fields serving as a pointer to the next structured object; in this case NEXT must advance IP by extracting the linkage information from the structured intermediate object and loading it into IP.

Note that in order for the interpreter to be able to function, NEXT must know about the details of the data structuring of the intermediate code objects; at the very least NEXT must know how to use a value in IP to extract an intermediate operation code and a new value for IP. The fact that NEXT is designed to interpret a specific type of structure means that the intermediate target machine provides a powerful, involved target for the compiler writer; target machine op codes need not be restricted to the conventional sequential machine word variety, and indeed intermediate code objects may represent complicated information structures for which the executable component is only one portion of the structure.

The second routine in the interpreter's collection is CALL. CALL's job is to initiate execution of the code sequence indicated by the operation code fetched by NEXT

in the preceding step. By definition a threaded op code is a pointer; that pointer may point directly to the op code's equivalent routine, or that pointer may be an indirect indicator used by CALL to find the actual address of the routine. In any event the first function performed by CALL is to take the intermediate operation code supplied by NEXT and transform it into a pointer to a routine used to support that intermediate op code. CALL's second action is to start execution of the indicated routine. If the routine is a primitive then a jump to subroutine instruction at the machine level is executed; provisions must be made for saving PC on the return address stack - this stacking mechanism is usually supported on the underlying machine. When the primitive routine completes execution it will execute a machine code level return from subroutine instruction - operation RETURN as supported at the machine code level - and the interpreter will resume execution by using NEXT to fetch the next intermediate op code.

CALL's job of initiating execution of the underlying routine becomes more involved when the indicated routine is a secondary, a sequence of intermediate code instructions. Conceptually CALL must call the complete interpreter recursively, using the secondary routine's address computed by CALL as an argument to be loaded by the recursively called interpreter into its local IP. The performance penalties which accompany actual use of recursion need

not actually be incurred. Instead CALL can push the current value of IP onto the return stack, load the secondary code address into IP, and allow NEXT to begin the subsequent fetch-execute cycle; NEXT begins interpretation of the called secondary.

In order for the above mechanism to work there must be a way whereby the interpreter can detect the end of a secondary subroutine, allowing the interpreter to pop the old, pre-call value of IP from the stack; RETURN performs this function. RETURN must pop IP for a secondary and PC for a primitive. The simplest way to detect the end of a secondary is to store a special sentinal value at the end of a sequence of intermediate op codes; when CALL detects this sentinal value it does not use the value to determine a routine address, but rather causes RETURN's instructions to be executed, thereby popping the stacked IP.

An alternative method is to implement RETURN as a primitive and generate an intermediate op code for this primitive at the end of each secondary sequence. The problem with this approach is that when the RETURN primitive is called it's return address (an address back in calling routine CALL) will be placed on the stack above the stacked IP which RETURN is to pop; RETURN would be required to swap these two items on the stack, an operation which consumes extra time for every secondary

RETURN executed. Another approach would be to never call intermediate-supporting primitives as subroutines, but rather jump to them without saving a return address. The machine code sequence which always follows execution of any primitive is execution of NEXT, so primitives merely GOTO NEXT; this method is used by FORTH.[21] This technique removes the necessity of placing a GOTO NEXT instruction at the end of the CALL portion of the fetch-execute sequence of the interpreter for primitives. RETURN for a secondary is implemented as an intermediate op code supported by a primitive; that primitive pops IP from the stack and jumps to the beginning of the interpreter at NEXT. Since the primitives do not require use of the RETURN stack there is no conflict in stack use between secondaries and primitives used as part of the interpreter.

These two alternative interpretive techniques are illustrated in Figures 1 and 2 on pages 26 and 27. The second algorithm eliminates the test for the sentinal op code required by CALL in the first algorithm; the second algorithm also eliminates the overhead of saving a return address on the stack when calling primitives. While these savings might seem to be slight at first thought, it must be remembered that the interpreter is

---

21. See C.H. Ting, Systems Guide to FIG-FORTH (San Mateo, CA.: Offete Enterprises, 1980), Chapter 4.

```
NEXT:    1. Extract an intermediate op code from
             intermediate data structure using
             value in instruction pointer (IP) and
             knowledge of organization of the
             intermediate data structure.
         2. Advance IP to point to the next inter-
             mediate data structure using current
             IP value and knowledge of intermediate
             data structure.
         3. Continue to step 1 of CALL.
CALL:    1. Examine op code extracted by NEXT.
             IF op code is secondary sentinal value
                 Pop IP from return stack
             ELSE
         2.      Translate intermediate op code into
                 a pointer to a routine, either a
                 primitive or secondary, which supports
                 that op code.
                 IF routine is a primitive
                     Call that routine, saving machine
                     PC on return stack.
                 ELSE
                     Push IP to return stack, then
                     load secondary routine address
                     into IP.
             3. GOTO NEXT
```

Primitive: A sequence of machine code instructions which may include calls to other machine code subroutines. The sequence is terminated by a return from subroutine instruction which pops return address in CALL from return stack and into PC.

Secondary: A sequence of intermediate data structures containing or otherwise referencing intermediate op codes. Intermediate op codes may be supported by either primitives or secondaries; care must be taken to ensure that no op code is supported by a secondary that unconditionally uses that op code, otherwise infinite looping will result, terminated by return stack overflow.
Sequence is terminated by SENTINAL value detected in CALL.

FIGURE 1

Interpreter mechanism with RETURN for primitives supported by popping program counter from stack, RETURN for secondaries supported by SENTINAL op code detected in CALL.

NEXT:

1. Extract an intermediate op code from intermediate data structure using value in instruction pointer (IP) and knowledge of organization of the intermediate data structure.
2. Advance IP to point to the next intermediate data structure using current IP value and knowledge of intermediate data structure.
3. Continue to step 1 of CALL.

CALL:

1. Translate intermediate op code into a pointer to a routine, either a primitive or a secondary, which supports that op code.
   IF routine is a primitive
       Jump to that routine; do not save PC on return stack.
   ELSE
       Push IP to return stack, then load secondary routine address into IP.
       GOTO NEXT.

Primitive: A sequence of machine code instructions which may include calls to other machine code subroutines. The sequence is terminated by a machine level jump to NEXT.

Secondary: A sequence of intermediate data structures containing or otherwise referencing intermediate op codes. Intermediate op codes may be supported by either primitives or secondaries, subject to same restrictions for secondaries given in Figure 1 on previous page. Sequence is terminated by op code for primitive RETURN code sequence

RETURN: A primitive which pops value of IP from return stack; value was pushed in CALL. Pop is followed by standard primitive jump to NEXT.

FIGURE 2

Interpreter mechanism with RETURN for primitives supported by a jump to NEXT, RETURN for secondaries supported by a primitive code sequence.

the most frequently executed piece of machine code in the entire interpretive system; savings of a few machine cycles will be multiplied many times. The second algorithm is not as modular as the first; the interpreter is broken up into the NEXT/CALL routine and the RETURN primitive, and op code primitives use GOTOs in order to return to the interpretation mechanism. Yet the second technique is orderly and highly structured in its own way, and it approaches the most efficient way in which to interpret a threaded intermediate code.

Arguments can be passed by a threaded calling routine to called primitive or secondary routines in a number of ways. Data values placed in-line in the threaded code, following the op code which directs execution to the called primitive or secondary, can be extracted by the called routine via use of the caller's IP. A called primitive must retrieve the value indirectly using IP as a data pointer, then advance IP to point to the next threaded op code following the last data argument. A called secondary would pop the caller's IP from the stack into a temporary pointer location, use that pointer to obtain the data, advance the pointer and push it to the stack to be popped into the caller's IP upon execution of RETURN. This in-line method of passing data can be used with all varieties of threaded code to be discussed.

Threaded jump instructions work in a similar way.

A jump primitive merely modifies the caller's IP; the threaded code interpreter automatically takes care of resuming the fetch-execute cycle at the new location.

Arguments can also be passed to called routines via a data stack, a stack distinct from that used to hold return addresses; this is the method used by FORTH.[22] A method could also be devised where return addresses, arguments, and local variables are grouped within stack frames, a scheme used for many fully compiled high-level languages.

Now that generic threaded code and its generic interpretation mechanisms have been seen, it is time to look at some of the detailed ways in which code threading is actually performed. Four varieties of threaded code have been identified; they have been labelled subroutine-, direct-, indirect-, and token-threaded code. These varieties differ in the ways in which they use pointers to access executable primitives.

> "Subroutine-threaded code: A sequence of subroutine calls with no other embedded instructions implements an intermediate language. Each subroutine call may be considered a single intermediate-language operation, which need not be related to the underlying machine architecture. Subroutine-threaded code (STC) is a control mechanism that is widely supported at the machine-hardware level."[23]

---

22. C.H. Ting, op. cit., Chapter 2, p. 5.

23. Terry Ritter and Gregory Walker, op. cit., p. 211.

The technique of supporting a virtual machine by creating a library of subroutines called in a conventional manner is a technique utilized by most programmers. Subroutine-threaded code represents an extension of the machine code level rather than a departure from it. Microcode still supplies the run-time interpretation mechanism; consequently this type of threaded code executes more quickly than the more machine independent varieties. Many traditional machine code compilers use system calls and collections of calls to library subroutines for performing special purpose operations such as input/output or database manipulation; the C language does not contain any I/O instructions and performs calls to I/O functions whenever input or output operations must be performed.[24] Subroutine-threaded code represents a slight departure from generation of pure in-line code by a compiler, a departure which does not require the services of a special run-time interpreter.

> "Direct-threaded code: Direct-threaded code (DTC) may be considered a sequence of machine-language subroutine calls with the 'call' op code removed. This results in a list of addresses, each of which points to a machine-language subroutine. Since the direct-threaded program includes no op codes, a short machine-language program must be written to read the next address in the list and transfer control to that

---

24. See Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language (Englewood Cliffs, N.J.: Prentice-Hall, 1978), p. 143-178.

> address. Traditional direct-threaded code implementations do not allow the use of true subroutines at the machine level but instead require that each routine terminate by executing the NEXT operation."[25]

Unlike subroutine-threaded code, direct-threaded code cannot rely on the microcode to provide the entire interpretation mechanism. The called routine cannot return to the calling routine since the latter does not contain executable operations. Instead control must be maintained by an interpreter which uses IP to fetch direct pointers to subroutines; CALL uses the pointers to provide direct jump addresses. Primitives can return using either of the two RETURN mechanisms previously described. Secondaries can be called from secondaries via manipulation of IP.

Obviously direct-threaded code is only a slight extension to subroutine-threaded code. Direct-threaded code in a sense supports a virtual machine more remote from machine code than that supported by subroutine-threaded code since direct-threaded code requires the services of a simple machine code interpreter. On most machines the only advantage to using direct- over subroutine-threaded code is that the former consumes less memory; call op codes can be assumed and do not need to be represented. On a microcomputer with eight bit op codes and sixteen bit addresses direct-threaded code

---

25. Terry Ritter and Gregory Walker, op. cit., p. 212.

routines require two-thirds of the storage required by subroutine-threaded code. Direct-threaded coding eliminates the possibility of interleaving threaded code calls with native machine code existant with subroutine-threaded code; direct threaded code must separate instruction sequences into distinct primitives and secondaries.

On some computers direct-threaded code will execute more slowly than the subroutine variety because of the requirement for an interpretation mechanism. Calls must be executed and IP must be managed, a two-step operation on most machines; subroutine-threaded calls make use of microcode support to manage jumps and return addresses using single instructions. A definite reduction in speed will be seen, however, if the RETURN mechanism of jumping from primitives back to the interpreter is used instead of subroutine calls to primitives; subroutine calls require the saving of a return address on the return stack, an operation which takes time. An intrasegment jump using an address in a register requires eleven clock cycles to execute on the 8086 microprocessor; the equivalent subroutine call requires sixteen.[26] The difference is caused by the lack of need to save the return address of the interpreter on the stack; subroutine-threaded code always saves the return address.

26. See Russell Rector and George Alexy, The 8086 Book (Berkeley, CA.: Osborne/McGraw-Hill, 1980), Chapter 3, p. 82 and 146.

An even greater savings in time resulting from the use of direct- rather than subroutine-threaded code can occur on machines which allow jumping indirectly via a register address (that is, use the address in the register to fetch the jump address from memory) and post-incrementing that register (assuming the register is being used as IP) within the same instruction; the PDP-11 is such a machine.

> "In comparing threaded code and subroutines for the specific PDP-11 example it can also be noted that the subroutine call occupies 2 words of storage while the 'indirect jump/increment register' instruction requires only a single word for address specification. Finally, execution of the latter instruction requires less than half the time required by the subroutine call/return sequence." [27]

Thus properly supported direct-threaded code can provide memory space and execution time savings when used instead of subroutine-threaded code. Time savings occur in the mechanism used to call primitives; nesting through several levels of secondaries can be very time consuming using any threaded code technique. Time consumption in these latter cases is primarily a function of the number of levels of secondary code through which the interpreter must search for primitives to execute.

A stated requirement of direct-threaded code is that it must point to an executable primitive; direct-threaded

---

27. Richard H. Eckhouse, Jr. and L. Robert Morris, Minicomputer Systems - Organization, Programming, and Applications (PDP-11) (Englewood Cliffs, N.J.: Prentice-Hall, 1979), p. 294.

code is, after all, a sequence of machine code subroutine calls with the op codes removed. How, then, is a call to a direct-threaded code secondary implemented?

The simplest way to handle calls to secondaries is to place a machine code primitive sequence before each secondary sequence; the duty of this machine code is to place the IP of the caller on the return stack, load IP with the address of the first memory location following the primitive sequence (the first direct-threaded op code), then jump to the interpreter. RETURN would be implemented using a primitive-supported direct-threaded op code to restore the caller's IP as the last instruction in the secondary. Consequently the main body of the interpreter would regard all support routines as primitives; secondaries would be prefaced by the primitive section which saves and modifies IP. This technique distributes the body of the interpreter throughout the code even more than the second generic technique shown on page 27.

> "Indirect-threaded code: Indirect-threaded code (ITC) consists of a list of addresses, but each address points to another address which then points to the machine-code routine. As compared to direct-threaded code, in indirect-threaded code the interpreter must go through an extra level of indirection. Indirect-threaded intermediate-language subroutines do not contain machine-language code for the call operation, and one advantage of indirect-threaded code is that a compiler using it need only produce pointers. By manipulating only pointers, the compiler generates intermediate-language code that does not include machine-language code itself; thus it is independent of the target

> machine. However, a disadvantage of indirect-threaded is that the interpreter has the overhead of an extra level of indirect addressing." [28]

In its simplest form indirect-threaded code consists of a string of addresses, each of which points to a memory location where there resides a pointer to an executable routine. I will call this type of code "simple indirect-threaded code." More complex indirection structures include intermediate codes which are not stored contiguously but rather as linked data structures; code links can be followed through more than the two levels of indirection mentioned above before reaching executable code. Indirect-threaded code is so called because it makes use of indirection in pointing to code; simple indirect-threaded code is the simplest way to implement such a code structure. Most of the following discussion of indirect-threaded code will deal with the simple variety.

Consider the generic intermediate code of page 27 and how it would be implemented using indirect-threaded code. The interpreter must take an indirect-threaded op code, obtain the contents of the memory cell to which it points, and jump to the location specified in that memory cell.

Primitives must be preceded by a memory location

---

28. Terry Ritter and Gregory Walker, op. cit., p. 212.

containing a pointer to the first executable primitive instruction. An indirect op code supported by a primitive consists of a pointer to that preceding pointer memory location. Primitives can return by jumping to NEXT in the interpreter.

Secondaries must also be preceded by a pointer to an executable code routine; this pointer could be called the code pointer prefix. The duty of the executable code routine is to save IP on the return stack, then load IP with the address of the first memory location beyond the location containing the prefix (the address of the first indirect op code in the secondary), then jump to NEXT in the interpreter. Secondaries are concluded with indirect pointers to a primitive which restores the stacked IP. It was previously stated that the machine code prefix for a called secondary of direct-threaded code is placed at the head of the called secondary; this requirement was necessary since the addresses indicated by direct threaded code pointers must be executable. Since the addresses indicated by indirect-threaded code pointers contain pointers to executable code, the entire prefix machine code sequence required for secondaries can be replaced by a single pointer to a machine code routine which need be stored only once rather than at the start of every secondary.

Increasing the number of levels of indirection

results in: 1) increased interpretation time required to trace through the levels, 2) increased distance between threaded op codes and machine code architecture, and 3) increased opportunities to embed threaded op codes and other pointers in complex data structures, making the virtual machine more complicated and powerful.

Since many possible indirect-threaded virtual machines could be designed, the comparison of indirect- to direct-threaded coding techniques is not as straightforward as the comparison of direct- to subroutine-threaded code. The major tradeoff involved in designing indirect-threaded machines is speed versus architectural complexity. The FORTH virtual machine which will be examined is a simple indirect-threaded virtual machine.[29] FORTH is known for the compactness of its code and speed of interpretation. The LISP indirect-threaded virtual machine is much more complex than that of FORTH; LISP is known for its power and flexibilty of data structuring, but it is definitely not known as a language which is efficiently executed by interpretation on a conventional machine level architecture.[30]

The fact that the three types of threaded code just

---

29. C.H. Ting, op. cit., Chapter 4.

30. See Bawden, Greenblatt, Holloway, Knight, Moon and Weinreb, op. cit., p. 345-373.

discussed all use machine address pointers limits the portability of these types of code. In order to execute threaded code produced for one computer on a different computer it is necessary that all called primitives and secondaries be at the same memory locations on both machines. This is possible when the two computers are the same make and model and are running identical interpreters with identical support routines. If two different types of computers are used then portability of the threaded object code would be very difficult to achieve. The run-time interpreters will not be identical on the two machines; algorithmically equivalent primitives will not require identical amounts of memory on both machines. If sizes of primitives differ then addresses will differ. In such cases portability will be possible only at the source code level; the user will be required to compile the threaded code for a particular computer and corresponding run-time interpreter, usually the machine on which the compiler is executed (unless a cross-compiler is used). The fourth type of threaded intermediate code, token-threaded code, does not suffer from this portability restriction.

> "Token-threaded code: The varieties of threaded code previously mentioned contained pointers that were actual addresses of the subroutines in memory. Using memory addresses to select routines wastes storage because the number of subroutines in the system is far smaller than the number of memory locations.

> A savings in intermediate-language program size can be obtained by using short tokens to identify the subroutines to be invoked. Typically, token-threaded code (TTC) can be implemented by using the current token to index into a table of subroutine addresses."[31]

Token-threaded code thus has two important advantages over the subroutine-, direct-, and indirect-threaded varieties discussed. Token-threaded op codes are more compact than the other varieties because they only use the number of bits necessary to distinguish between all possible operations in the virtual machine; the others use the number of bits necessary to distinguish between all possible memory locations in the program's address space. Token-threaded code's second advantage is portability of compiled code. If token-threaded code is used to index into a table of subroutine addresses then that table can be built into the run-time interpreter for each computer. Consequently operations are not bound to addresses at compile-time but only at run-time. If user created procedures are to be called there must exist some method for adding addresses of user defined secondaries to the token-indexed address table; token op code indexes must be assigned to user defined secondaries at compile-time, but the corresponding code addresses in the indexed table will not be resolved until load/run-time.

---

31. Terry Ritter and Gregory Walker, op. cit., p. 212.

Token-threaded intermediate code adds another level of indirection to that of indirect-threaded code; the pointers contained in the index table are normally some variety of indirect pointers. The user of token-threaded code trades the speed advantage of the more machine dependent varieties for portability of compiled code. Thus we see the primary advantage of using the fourth type of threaded code.

FORTH and LISP both generate code structures containing indirect pointers, pointers indicating actual memory addresses. The loss of portability is not too important, however, since both languages are geared toward interactive use; the emphasis of interactive languages is interaction with the programmer/user, making accessibility of source code information important to the virtual machine (especially for the support of intelligible error messages and testing/debugging facilities). Portability at the source code level can be supported for both languages. Since both of these languages place great emphasis on user interaction it is important to see in what ways threaded code can be used to support interactive programming language facilities.

INTERACTIVE THREADED LANGUAGE FACILITIES

Threaded language facilities are not inherently interactive. Threaded code generating compilers for non-interactive languages such as FORTRAN IV have been designed for widely used computers.[32] Yet the high level virtual machine created by an intermediate code has several qualities which allow it to readily support an interactive programming environment.

An interactive language system aids the programmer at program text entry time by reporting syntax errors as soon as they occur. Such reporting is possible because a text editor geared to the specific language is used; the editor is often written as part of the language system. With incremental compilation techniques routines can be compiled as they are entered and executed independently of the main program; this ability is useful for subroutine testing, and is possible with both FORTH and LISP. The text editor in such a system not only stores source text in a file, but also passes text to the incremental compiler for syntax verification and compilation.

---

32. Richard H. Eckhouse and L. Robert Morris, op. cit., p. 294.

"Interactive languages have led to the development of a new type of compiler, called an incremental compiler. An incremental compiler takes the source program one line at a time. It checks each line for errors and translates the line into the internal language. If the line is correct, its translation is then incorporated as an 'increment' to the internal program; otherwise the error is reported to the user and the line is ignored." [33]

A complete function definition in the two languages studied is compiled as soon as the function's text is read by the language system; function definition text can be read either from a keyboard or a disk file with these two languages - the two sources of text are treated as equivalent. Procedures in both languages are treated as independent functions; the "main program" is just another function which invokes execution of the other functions making up the program. Individual functions can be executed interactively by the user as though they were the complete main program.

The threaded code virtual machine supports the interactive mechanism of incremental compilation because the virtual machine interpreter is present in the computer as the programmer is entering source text; compilation of the source code in both languages studied is simplified because of the simple syntax used by FORTH and LISP. Each instruction entered by the user

---

33. P. J. Brown, op. cit., p. 41.

corresponds to one operation code of the virtual machine. The compiler matches operations specified in the source text to underlying primitive and secondary supporting routines in a one-to-one correspondence. Threaded op codes can be generated in sequence as commands are read. If an error is discovered then it is reported and the function currently being defined is discarded; if there are no errors then prefixes and suffixes can be attached to the function definition if necessary, effectively extending the virtual machine by creating a new secondary which can be called by other procedures or called by the user from the console.

Forward references to procedures not yet defined are treated differently by different languages. FORTH allows no forward references, so a command can be bound to its supporting routine as soon as the command is recognized. LISP allows forward references, creating a partial intermediate representation for an undefined function reference but marking it as undefined until function definition time; conceptually an address is reserved for the indirect pointer to the function, so that while the address of this pointer can be resolved when the undefined reference is made, the contents of the pointer (address of the routine) must be filled in when the function is defined. Obviously a function cannot be executed, interactively or otherwise, unless

the functions which it calls have been defined.

Extensibility at the source level is true of both FORTH and LISP because function/procedure invokations in these languages use the same syntax as invocations of the basic commands of the languages; essentially all commands are written as function calls. This source level extensibility meshes nicely with the virtual machine level extensibility - whenever a new primitive or secondary routine is defined by the user the address of that routine becomes a new threaded op code available to the compilation mechanism. Thus programmers using these languages can extend the capabilities of the virtual machine, a condition which is desirable when the virtual machine is being used to experiment with novel language features or machine architectures.

The fact that threaded code systems use both a compiler and an interpreter has already been established; the compiler generates threaded op codes at compile-time and the interpreter executes them at run-time. Interactive code systems also make use of another interpreter, the command interpreter.[34] The duty of the command interpreter is to evaluate user commands read either from a terminal or a disk file. Two modes of operation are

---

34. See R. G. Loeliger, op. cit., p. 9-38 for an overview of the interactions of the command interpreter, run-time interpreter and threaded code compiler.

possible in such a system - interactive execution mode and compile mode. In interactive execution mode the command interpreter takes a single user supplied text command, sends it to the compiler for function address resolution, then initiates CALL to start execution of that function immediately; the command interpreter implements NEXT by reading the next text command and performing another compile-and-execute sequence.

When an immediate command to enter compile mode is recognized by the command interpreter then the workings of that mode begin. Space is allocated for the definition of a new routine and the source text which makes up its definition is read and compiled into that space. The compiler builds any intermediate data structures required by function definition (such as supplying code prefixes, suffixes, and any intermediate linkage mechanisms needed by the virtual machine). Instructions representing compiler directives are not compiled but are executed immediately; this class of instructions includes the command to complete definition of the function and return to immediate execution mode. Of course it may be possible to specify explicitly that a compiler directive be compiled as part of the function definition; this specification might be made if the compiler itself were being extended by the definition of a new compiler directive. Such extensions are possible in an interactive interpreted system

because the compiler as well as the command and threaded code interpreters (sometimes called the outer and inner interpreters respectively) are all present in memory when the language system is being used.

Threaded code systems thus support incremental compilation and immediate execution of commands because resolution of thread addresses can be performed at program entry time; support routines are already resident in memory. Threaded code systems also support interactive program testing and debugging because fields can be attached to the intermediate code data structures to store source code pertinent information such as variable names, function names and breakpoints. The inner interpreter can be designed to test the breakpoint field, entering an interactive debugging routine when a breakpoint is detected; such a routine could also be entered when the inner interpreter detects a virtual machine error such as division by zero, data type mismatches, or call to an undefined function.

In fact it is possible to allow the programmer to specify whether the compiler and inner interpreter will operate in the debug mode. Debug mode compilation would build debugging information into the intermediate data structure for the debug mode interpreter to utilize; non-debug mode compilation would not store this information, allowing faster compilation and interpretation.

Since the virtual machine architecture is geared to a specific programming language it is possible to build machine operations to support interactive facilities such as special terminal handlers and editors into the virtual machine. If the virtual machine is extensible then programmers can customize the machine to provide a very powerful user interface.

Just as the basic functions supported by a threaded code virtual machine code can in theory be supported by machine code level computing without the use of an intermediate code, so could interactive extensions be so supported. The advantage to using an intermediate code virtual machine to support an interactive programming environment is again the ease with which operations can be specified for execution by a high-level machine; language facilities which in theory could be specified at machine code level might in fact never be written because of the complexity of such machine code. An extensible virtual machine allows language facilities to be added incrementally as they are desired. If at some later time a microcoded interpreter is written to support the virtual computer at machine code level then the interactive facilities can be made part of the computer's basic architecture. Such a trend can be seen in the design of stand-alone special purpose interactive computing equipment such as microcomputer-based

workstations.[35]

THE FORTH PROGRAMMING LANGUAGE[36]

FORTH is an interactive, extensible language that was originally designed to monitor and control radio astronomy equipment.[37] The language's most popular use today is in the monitoring and control of laboratory and industrial equipment. The simple indirect-threaded code upon which the FORTH system is built corresponds very closely to the intermediate code structure associated with the generic interpreter mechanism of Figure 2, page 27. FORTH threaded code secondaries are stored as contiguous sequences of indirect machine address pointers, but the extensibility of FORTH which extends to the compilation

---

35. See Harvey H. Hindin, "Revolution Brewing in Workstation Technology," Computer Design, Vol. 24, No. 1 (January, 1985), p. 111-124.

36. In the interest of brevity the use of the FORTH and LISP languages at the source code level will not be examined in detail; instead the discussion will be oriented toward features of those languages supported at the threaded intermediate code level. For information on FORTH programming see E. Rather, L. Brodie and C. Rosenberg, Using FORTH (Hermosa Beach, CA.: FORTH, Inc., 1980). For information on LISP programming see Patrick Henry Winston and Berthold Klaus Paul Horn, LISP (Reading, MA.: Addison-Wesley, 1981).

37. See Charles H. Moore, "The Evolution of FORTH, an Unusual Language," BYTE, Vol. 5, No. 8 (August, 1980), p. 76-90.

and interpretation mechanisms allows for the creation of automatic support facilities for more complex information structuring schemes.

FORTH intermediate op codes are supported by primitive or secondary routines with the following structure:[38]

Location:	Contents:
Name Field Address (NFA):	Routine Header
Code Field Address (CFA):	Executable Code Pointer
Parameter Field Address (PFA):	Body of the Routine

The header contains information needed by the compiler and command interpreter; its structure will be discussed shortly.

The executable code pointer is located at a place in the routine known as the code field address (CFA); this is a direct pointer to the initial executable machine code for the routine. The CFA of a routine is the value contained in the intermediate op code supported by that routine. For primitives the CFA contains the address of the first executable machine instruction in the body of the routine; this address is the parameter field address (PFA) for primitives. Thus the CFA of a primitive always points to the next machine word following the CFA; the primitive routine is terminated by execution of the NEXT routine which advances the

---

38. See W. F. Ragsdale, op. cit., p. 4.

instruction pointer and jumps to the start of the interpreter fetch-execute cycle.

The CFA of a secondary in FORTH points to a routine called "DOCOL" which stands for "do colon." This name refers to the fact that the colon symbol ":" is used to initiate compilation of a secondary in FORTH source notation. DOCOL performs the task of pushing the IP to the return stack and loading IP with the address of the first intermediate op code of the routine; the body of the routine consists of a sequence of indirect threaded op codes which extends from the PFA of the routine through the last instruction. The last instruction is an indirect pointer to the primitive routine "SEMIS," a routine named after the source semicolon directive ";" used to complete the source definition of FORTH secondaries. SEMIS completes interpretation of a secondary by popping the calling routine's IP value from the return stack; SEMIS is the RETURN primitive of FORTH.

Information in the routine header at the name field address (NFA) is used by the compiler to locate FORTH routines.[39] The primitive and secondary routines of FORTH are compiled in a last-in/first-out format into a data structure known as the "dictionary." The dictionary

---

39. A most thorough examination of the workings of the components of the FORTH intermediate code compiler is contained in C.H. Ting, op. cit.

grows from low to high memory as definitions are compiled with each new definition compiled just beyond the last. The "FORGET" command, when given the address of a compiled routine, will erase definitions from that of the most recently defined (highest in memory) through that of the addressed routine; forgetting is accomplished by changing the address contained in the top-of-dictionary pointer from the address of the most recently compiled routine to the address of the routine immediately preceding the forgotten one.

Routine headers are tied together in a linked list fashion; the list extends from the most recent definition at the top of the dictionary stack to the first definition at the bottom of the dictionary. The first field in the header is an encoded byte that contains the length of the routine name in the lower bits. This field is followed by the actual ASCII string representing the name of the routine. After the name comes the link field which contains a pointer to the previously defined routine in the dictionary.

The compiler translates routine names read from the user's console or a source file to code field addresses by searching the dictionary linked list sequentially from the most recently defined routine; name fields are compared to source text commands, with the name length field used to quickly eliminate name fields not equal in

length to the source command. When the correct source-matching name field is found then the code field address of that routine is returned. If the interpretive system is in immediate execution mode then the returned code field address is loaded into a jump register as though it had been fetched by IP from a secondary and an indirect jump to the code field address takes place, executing the routine. If the system is in compile mode then the returned code field address is stored into the secondary routine currently being formed by the compiler. An exception to this treatment of the returned address in compile mode occurs if the command is a compiler directive which is to be executed immediately in this mode; such immediate routines are identified by an immediate flag bit encoded into the first byte (length byte) of the routine header. When the immediate bit is set the code address is given by the compiler to the inner interpreter for immediate execution in compile mode. An additional bit in the first byte of the header, the "smudge bit," is toggled at completion of compilation of the routine to show that the definition of a valid threaded code sequence is complete; should an error occur during compilation the compiler reports the error immediately to the console, keeps the smudge bit in its "dirty" state so that the partially defined routine will be identified as invalid by future scans of the compiler (the incomplete

definition is not actually removed from the dictionary), and the compiler returns immediately to the interactive command interpreter state.

This simple compilation technique of following the linked list of routine definitions and examining names is possible in part because the syntax of FORTH source code is so simple. FORTH makes use of two run-time data stacks, the return stack used for storage of nested IP and program counter return addresses and the user data stack used for passing data values among called routines. The FORTH virtual machine is a stack machine which uses the data stack for all local data manipulation; most operations pop their input arguments from this stack and push their results to it. Only global variables, variables which are stored statically in the dictionary, are actually referenced by name, and even these must have their values or addresses placed onto the data stack before they are accessible by most routines. FORTH source code is written in postfix notation with reference to the data stack implicit in the operation of instructions which use the stack. Therefore compilation consists merely of converting sequences of postfix source commands into sequences of equivalent indirect op codes and storing these in the dictionary.

Compilation of flow-of-control constructs is only slightly more involved. The "IF", "THEN", "WHILE",

"REPEAT", "DO", "LOOP", and associated commands are treated as compiler directives which are used to compile one of two intermediate operations: "BRANCH" or "OBRANCH". These latter primitives are used respectively to modify the current value of IP unconditionally or when the value on top of the data stack is zero; they implement unconditional and conditional jumps at the intermediate code level. No technique for specifying program text labels is supported at the FORTH source level, so BRANCH and OBRANCH cannot be readily used as go to instructions by the FORTH programmer; structured flow-of-control constructs are the control method supported for FORTH applications programming. Since the "IF", "WHILE", and associated commands execute as compiler directives at compile time, they can make use of the data stack to push and pop flag values to ensure that correct nesting of control constructs takes place; compile-time use of the data stack does not interfere in any way with run-time use of the stack by code being generated by the compiler. For instance, IF pushes a flag onto the data stack, ELSE (which is optional) tests for this flag and ENDIF tests for and pops this flag. DO pushes a different flag for which LOOP tests and which LOOP pops. If a DO-LOOP is nested within an IF-ELSE construct then the DO-LOOP nesting flag will have been popped from the stack by LOOP before ENDIF tests for the IF flag. Correct nesting of

structured control commands is thus assured. Addresses of the operand fields for compiled BRANCH and OBRANCH operations can also be passed among these compiler directives on the data stack at compile time, allowing forward branches from IF, ELSE, and DO commands to be filled in by corresponding ELSE, ENDIF, and LOOP commands when the destination address of the branch is reached by the compiler. Much of the work of compilation is embodied in the definition of compiler directives such as IF and DO. FORTH users can create compiler directives (i.e. specify that the command being compiled have the immediate bit in its header set for compile-time execution) by following the source definition of the command with execution of the IMMEDIATE command. Thus the FORTH compilation process is extensible. An example of a possible user extension to the standard FORTH control constructs will be given in a few pages.

The NEXT, CALL, and RETURN mechanisms of FORTH's inner interpreter correspond very closely to the generic mechanisms developed previously in this essay. As already stated RETURN is implemented as a jump to NEXT in the interpreter for primitives and as primitive SEMIS for secondaries.

In addition to the data stack, return stack, and instruction pointer for intermediate code (IP), FORTH's inner interpreter uses an executable code pointer called

the current word pointer (abbreviated "W" in most FORTH systems); registers are used for IP and W in PDP-11 FORTH. The PDP-11 implementation of NEXT and CALL is:[40]

```
NEXT:       MOV (IP)+,W
            JMP @(W)+
```

The MOV instruction fetches the indirect-threaded op code indicated by IP and advances IP to point to the following op code. The doubly indirect JMP command takes the contents of W (the indirect-threaded op code) as a pointer to an address and the contents of that address as a pointer to the address to which control is transferred. After obtaining the jump address W is incremented; in the case of jumping to a secondary (the actual jump is to the DOCOL routine whose address is in the code field of the called secondary) this leaves W pointing to the first intermediate op code of the called secondary. DOCOL's actions amount to a simple sequence:

```
            MOV (IP),-(RP)      ; stack caller's IP,
            MOV W,IP            ; start of called
            NEXT                ; secondary routine
```

The first MOV pushes the caller's IP onto the return stack. The second MOV loads IP with the pointer to the called secondary sequence. The two-instruction NEXT for

---

40. See John S. James, op. cit., for PDP-11 FORTH source information.

the PDP-11 is encoded as a macro which is expanded at the end of every primitive code sequence; here NEXT initiates interpretation of the called secondary.

The SEMIS routine which implements RETURN for secondaries is also a very compact sequence:

```
MOV (RP)+,IP
NEXT
```

The stacked IP value of the calling routine is popped into IP and NEXT is executed to resume execution of that routine.

The heart of the FORTH indirect-threaded code interpreter is thus implemented in five machine instructions on the PDP-11. Such economy of code requirements is possible in part because, "The PDP-11 instruction set is very close to what is required optimally to implement a virtual FORTH computer."[41] Eight-bit microcomputers normally do not have central processor registers available for dedication to inner interpreter register needs and must fetch IP and W from and store these registers to assigned memory locations. Such microcomputers also generally do not have available all of the addressing modes and register post-incrementing options available with the PDP-11 and must use several instructions to do what the PDP-11 can do with one. For this reason NEXT is generally written

---

41. C.H.Ting, op. cit., Chapter 1, p. 2.

as a distinct routine on microcomputers; it is stored once and primitives jump to NEXT, requiring the extra execution time involved in a jump in order to avoid expanding a multiple-instruction NEXT sequence at the end of every primitive. Nevertheless FORTH still generates economic intermediate code sequences on eight-bit microcomputers, and its interpretation mechanism is sufficiently simple so that a large percentage of execution time can be spent executing the primitives of interest rather than searching them out with the inner interpreter. The modest requirements that FORTH places on the host computer, due to both modesty of threaded code memory requirements and simplicity of postfix source code compilation, in combination with the relative speed of the interpretation mechanism make it a language highly suited to implementation on small microcomputer systems. Notable examples of such systems include dedicated processors used to monitor and control scientific and industrial equipment.

An aspect of FORTH which helped to popularize the language is its extensibility. At least three types of FORTH extensibility have been identified:

"Level I: Using standard FORTH defining words to add new operations (programs).

Level II: Creating new user-defined defining words that, in turn, create new classes

of words.

Level III: Creating new FORTH-like systems through metaFORTH."[42]

The first type of extensibility is due to two aspects of FORTH discussed in the section on interactive threaded systems. Since built-in FORTH primitives and secondaries and user-defined primitives and secondaries are invoked using identical source notation, that is as one-word function calls arranged in postfix order, the user-defined routines appear at the source level as extensions to the FORTH language; and since user-defined routines are compiled into code sequences which support indirect-threaded op codes, the user routines extend the FORTH virtual machine at the intermediate code level. Ordinary definition and compilation of functions in FORTH thus extend the language.

The third type of extensibility mentioned above is due at least partly to the ability of the FORTH programmer to define multiple vocabularies. It was previously explained how the basic dictionary of definitions is searched sequentially by the compiler by following the linked list implemented in the routine headers. It is

---

42. Kim Harris, "FORTH Extensibility," BYTE, Vol. 5, No. 8 (August, 1980), p. 167.

possible for a FORTH programmer to define several different vocabularies, one of which may be active at a given time. These vocabularies represent separate, parallel linked lists of definitions which are linked at their tail ends to the basic FORTH definition linked list, but which are not linked to each other. When an alternate user-created vocabulary is activated, its linked list of definitions is searched by the compiler before the basic dictionary is searched; any FORTH command can be redefined in either the basic or in a user-defined vocabulary. It is thus possible to create an alternative vocabulary where the entire FORTH interpretation and compilation system is redefined; when this vocabulary is activated the programmer's environment becomes that of the alternative vocabulary. This use of alternate vocabularies in combination with the first two types of extensibility is what is meant by "creating new FORTH-like systems" above.

The second type of extensibility is in many ways the most interesting and the type which will be discussed here. What is meant by "creating new user-defined defining words that, in turn, create new classes of words," is the creation of compiler directives and, more importantly, creation of commands which have impact at both compile-time and run-time by creating new data and instruction creating and processing mechanisms.

As an exercise in creating compiler directives I decided to implement a simple case construct for FORTH; none is available in the systems which I installed. I desired a construct which would take an integer from the data stack and execute the corresponding command (in FORTH jargon the corresponding "word") in a list of commands within the case construct. For instance the case statement might appear as follows:

```
CASE
        COMMAND1
        COMMAND2
        COMMAND3
        COMMAND4
ENDCASE
```

One or more FORTH words can appear between CASE and ENDCASE. If this construct is reached at run-time with a value of three on the data stack then the third command in the list will be executed; execution will then resume beyond the ENDCASE statement. If the integer on the data stack is outside the range of the command list (less than one or greater than four here) then control will resume immediately beyond the ENDCASE statement.

The results of my efforts can be seen in Figure 3 on page 62. Since FORTH source code can be very cryptic for the uninitiated I will explain this code line by line.

```
1     : CASE
2                 ?COMP
3                 COMPILE BRANCH
4                 HERE
5                 Ø ,
6                 9 ; IMMEDIATE
7     : (ENDCASE)
8                 DUP  >R
9                 ROT 2 *
10                R>  +
11                DUP  >R
12                <  IF
13                            R >  IF
14                                        R @ EXECUTE
15                            ENDIF
16                ELSE
17                            DROP ENDIF
18                R>  DROP ;
19    : ENDCASE
20                ?COMP 9 ?PAIRS
21                DUP HERE DUP  >R
22                OVER - SWAP !
23                R>  COMPILE LIT ,
24                COMPILE LIT ,
25                COMPILE (ENDCASE) ; IMMEDIATE
```

FIGURE 3 - A FORTH Case Construct Definition

Three FORTH commands are defined here: CASE, (ENDCASE), and ENDCASE. The definition of a FORTH secondary extends from the colon which initiates compile mode to the semicolon which terminates it. (ENDCASE) is a run-time routine used to complete some of the compile-time actions initiated by ENDCASE. (ENDCASE) is defined before ENDCASE because FORTH does not allow forward references to commands; the valid definition of a command must be complete before that command can be used. I will reverse the order of explanation of these two routines because ENDCASE the compiler directive is executed before (ENDCASE) the run-time support routine.

The definition of CASE extends from line one through six. Note that the terminating semicolon is followed by the IMMEDIATE command, the command which forces the immediate bit in the header of CASE to be set; consequently CASE is treated as a compiler directive and its contents are executed rather than compiled when CASE is encountered at compile-time.

The first command in CASE, ?COMP on line two, tests to ensure that the system is in compile mode when CASE executes; ?COMP will print an error message and abort the execution of CASE if this condition is not met.

The COMPILE BRANCH sequence in line three forces the threaded op code for the BRANCH routine to be compiled into the current compilation spot in the dictionary.

Remember that since the CASE construct can only be activated in compile mode, the compiler will already have begun the storage of a secondary sequence into the dictionary (i.e. the colon for the user-defined secondary will already have been seen and a linked secondary code sequence will be under construction) when CASE begins execution; thus CASE compiles the threaded BRANCH instruction into a secondary sequence already begun. Placing COMPILE before BRANCH forces the BRANCH routine address to be compiled rather than executed, even though the commands within CASE normally execute at compile time; COMPILE modifies this compile-time immediacy for the command which follows it. COMPILE also advances the dictionary code pointer (where compilation is taking place) after storing the code address of the BRANCH primitive in the dictionary.

HERE on line four, which executes at compile-time, places the current compilation address in the dictionary (the first address after the BRANCH instruction) onto the data stack where it can be accessed by ENDCASE; line five, "Ø ,", pushes zero to the data stack then pops this zero and compiles it into the dictionary (the action of the comma operation), again advancing the compilation pointer. The HERE address - the address of the compiled zero - remains on the data stack. The zero is a place holder for the BRANCH instruction's offset; the zero will be over-written by ENDCASE with an actual BRANCH offset.

Line six pushes the value nine to the data stack at compile-time; we will see why shortly. The semicolon ends the definition, returning the system to immediate execution mode; the IMMEDIATE sets the compiler directive status of CASE.

It is important to track the status of two data structures when discussing compiler directives - the data stack and the code sequence being created in the dictionary. So far the code sequence looks like this:

Threaded operations compiled before executing case

.
.
.

BRANCH

Ø

The data stack looks like this (top of stack to the left):

(t.o.s.) - 9 branch-offset-address

CASE has completed execution at this time. Since we are in compile mode the compiler reads the next command, "COMMAND1" in our example on page 61. Assuming that these are ordinary commands (not compiler directives), the compiler will create a sequence of intermediate op codes, one for each command. After resolving addresses for the four commands the dictionary entry, starting with the BRANCH instruction of CASE, will look like this:

```
                BRANCH
OFFSET:         Ø
                op code for COMMAND1
                op code for COMMAND2
                op code for COMMAND3
                op code for COMMAND4
```

Now the compiler reaches ENDCASE; refer to its definition on page 62. The IMMEDIATE in line twenty-five shows that ENDCASE is a compiler directive, so unlike the four commands compiled prior to ENDCASE, ENDCASE is executed at once.

ENDCASE tests for compile mode with ?COMP. ENDCASE then pushes a nine onto the data stack and uses ?PAIRS to test whether the two numbers on top of the data stack are equal; if not then a compiler error is reported and compile mode is aborted. The pair of nines on the data stack is used to assure the compiler that ENDCASE is matched properly with CASE. In a similar fashion the IF-ELSE-ENDIF directives pass a two amongst themselves on the data stack and the DO-LOOP directives pass a three; absence of prerequisite directives or improper nesting of control constructs are detected in this way.

The address of the BRANCH address from CASE remains on the stack. At line twenty-one DUP duplicates this address on the stack. HERE then places the current address of compilation in the dictionary (the first

address beyond the compiled op code of COMMAND4) onto the data stack and that is duplicated. "To-R"(>R) moves the item on top of the data stack (the duplicated copy of the end of dictionary address) onto the return stack for temporary storage. At this point the return stack has a copy of the current compilation address, and the data stack looks like this:

(t.o.s.) - current-compilation-address branch-offset-address
branch-offset-address

In line twenty-two OVER makes another copy of the branch-offset-address above the current-compilation-address on the data stack. Minus ("-") subtracts the branch-offset-address from the current-compilation-address, popping these two values and leaving their difference on the data stack:

(t.o.s.) - distance-from-branch-to-current branch-offset-address

SWAP in line twenty-two swaps the top two data items and "!" stores the second item on the stack at the address specified at the top of stack. The effect is to replace the "BRANCH Ø" with a BRANCH to the first instruction following COMMAND4. All that remains on the data stack is another copy of the branch-offset-address.

At line twenty-three "From-R" (R>) returns the value saved on the return stack to the top of the data stack; this is the saved copy of the current compilation

address. COMPILE LIT compiles the literal command, a command for taking an in-line integer value in a secondary code sequence and pushing it to the data stack; at run-time when LIT executes it will take the word following its op code (accessed via IP) and push it to the data stack, advancing IP. The comma command in line twenty-three compiles the value on the top of stack at compile time (which is the address of this LIT instruction) into the address following LIT. When LIT is executed at run-time it will thus push its own address to the data stack; we will see why shortly.

At line twenty-four LIT is compiled again, followed by a comma to compile the top of the data stack into the location following LIT. That value is the final copy of the address of the offset of the branch instruction. The data stack is back to where it was at when CASE was encountered; all values pushed by CASE-ENDCASE have been used. Finally the op code for (ENDCASE) is compiled by ENDCASE to execute at run-time. The generated code sequence looks like this:

```
                BRANCH
OFFSET:         1Ø
                op code for COMMAND1
                op code for COMMAND2
                op code for COMMAND3
                op code for COMMAND4
```

```
RESUME:         LIT         ( push next word to data stack)
                address-of-RESUME
                LIT
                address-of-OFFSET
                op code for (ENDCASE)
```

Notice that the distance from OFFSET to RESUME is 1Ø (decimal). The PDP-11 is byte addressable, so that the five word distance equals ten byte addresses. The offset distance is computed using the compilation pointer and will be correct for the machine on which it is calculated.

Keeping the above code sequence in mind, consider what happens at run-time. Assume we have a three on the data stack when CASE is encountered, meaning that COMMAND3 should be executed. BRANCH 1Ø jumps over the commands within the CASE construct. The first LIT pushes the address of RESUME onto the data stack above three. The second LIT pushes the address of OFFSET, so when the (ENDCASE) op code is reached the data stack at run-time looks like this:

(t.o.s.) - address-of-OFFSET address-of-RESUME 3

Now (ENDCASE) executes; refer to page 62. Line eight duplicates the address of OFFSET and moves the copy to the return stack. Line nine rotates the three up from the third place to the top of stack; "2 *" multiplies it by two to make it a byte offset (six bytes) into the

sequence of CASE commands. From-R brings the address of OFFSET back from the return stack and "+" adds the six byte offset to it. Referring to our compiled code sequence on pages 68 and 69 we see that OFFSET+6 yields the address of the COMMAND3 op code. The IF tests on lines eleven through thirteen are used to ensure that the op code address falls between OFFSET and RESUME. If so then fetch ("@") retrieves the threaded op code for COMMAND3 from the CASE list and EXECUTE (line fourteen) calls the inner interpreter to stack the current IP and execute the routine whose code field address is on top of the data stack. Lines fifteen through eighteen complete the IF constructs and clean up the data stack.

This example has demonstrated several things. One is that postfix FORTH source code is terse and cryptic and easier for a computer to read than for a human. Another thing demonstrated is that, given familiarity with FORTH's internals, the control constructs of the language can be extended.

The second type of commands which implement the Level II extensibilty of FORTH of page 58 are those which build information structures at compile-time and compile processing facilities to execute them at run-time. These are even more involved than the compiler directives so I will not examine them in detail but will merely explain an example of such commands in general terms.

A pair of FORTH commands which create information structures and run-time handling facilities are the commands BUILDS and DOES. Suppose one wishes to extend FORTH by creating an array definition and handling facility since FORTH does not support arrays. We could define ARRAY as follows:

```
: ARRAY
        BUILDS
                Code to allocate array storage in
                the dictionary based on dimensions
                supplied on the data stack to ARRAY.
                Also dimensions are stored at head of
                array for run-time subscript error
                checking and address calculation.
        DOES
                Code to be executed when an instance of
                an array is referenced. This uses
                subscripts supplied on the data stack
                and dimensions stored by BUILDS above
                to check array bounds and compute the
                array element address
;
```

We might now define a two-dimensional array of size 3 x 3 called EXAMPLE in the following way:

```
3 3 2 ARRAY EXAMPLE
```

The two threes are the dimensions and the two tells

ARRAY how many dimensions to pull from the stack. When ARRAY is executed it creates a dictionary entry for EXAMPLE and executes the BUILDS code to build EXAMPLE by storing the dimension information and allocating memory for array EXAMPLE following the header for EXAMPLE in the dictionary. Note that BUILDS in effect creates some type of intermediate code data structure, possibly different from the simple indirect-threaded code upon which FORTH is based; the type of intermediate code structure BUILDS creates depends upon the FORTH programmer. Furthermore ARRAY links the code field of EXAMPLE to the code compiled after DOES; this is the code which will be run when EXAMPLE is invoked. Suppose the following command sequence is specified:

```
1 2 EXAMPLE
```

The above states that we wish to obtain the address of row 1, column 2 of ARRAY EXAMPLE; EXAMPLE will leave this address on the data stack. What actually happens is that the code sequence following DOES in the definition of defining word ARRAY takes these subscripts and compares them to the dimensions stored when ARRAY created EXAMPLE for validation. The DOES code also uses the subscripts obtained from the data stack and the stored dimensions to calculate the address of the referenced element, which DOES then leaves on the data stack. ARRAY's DOES code is able to locate EXAMPLE rather than some other array

because the DOES code of EXAMPLE can use the code field pointer for EXAMPLE to locate EXAMPLE's data at run-time. EXAMPLE appears as follows in the dictionary:

```
EXAMPLE header
code pointer to secondary code of ARRAY's DOES
actual array structure allocated by BUILDS
```

The above three sections of EXAMPLE are located at its name field address, code field address, and parameter field address respectively. The inner interpreter will nest down to the code field pointer location at run-time (the DOES code), using EXAMPLE's op code (pointer to code field address) to find the actual array.

Once EXAMPLE leaves the referenced element's address on the data stack the command following EXAMPLE can read from or write to that element's address.

An array builder is a simple example of the power of BUILDS-DOES. The point is that FORTH provides mechanisms for building alternative intermediate code strucures and interpreting them at run-time. Some or all of the commands in the BUILDS and DOES sections themselves might not be supported by simple intermediate code; these commands might have been built by earlier invocations of BUILDS-DOES. The simple intermediate code mechanism provides a base upon which to build FORTH programs, but it is not the only type of intermediate code mechanism which may be used.

In addition to supporting techniques for generating and interpreting complex types of intermediate code, most FORTH systems support postfix assemblers used to generate user-defined primitive sequences. Such sequences are useful for programming time-critical applications and essential for performing operations such as hardware interrupt trapping from FORTH.

The flexibility, compactness, and general speed of the FORTH virtual machine have gone a long way in offsetting the disadvantages of using its terse, cryptic postfix source code. FORTH's extensibility is to a large extent responsible for its continuing existence. In addition to its use in machine control and monitoring FORTH probably has its greatest utility as an exploratory, educational tool. FORTH provides valuable lessons to be remembered when designing new programming languages and especially when designing the virtual machines to support them; the lessons it provides concern extensibility.

THE LISP PROGRAMMING LANGUAGE

Like FORTH, LISP is an interactive, extensible language supported by a threaded intermediate code interpretation mechanism. LISP's intermediate machine representation is a form of complex indirect-threaded code. The simple threading of FORTH relates primarily to sequences of compiled FORTH procedural code to be interpreted at run-time, rather than with data acted upon by the executing procedures. It is possible to create complex threaded data representations using commands such as BUILDS-DOES in FORTH, but the basic language implementation treats data as the contents of simple words of the underlying hardware machine. FORTH's threaded structure is geared toward efficient interpretation of simple indirect-threaded op codes.

The internal threading scheme of LISP is oriented toward overall data structuring, with executable code sequences treated as components of more general, encompassing collections of information organized in the form of linked lists. The prefix notation of LISP source code, like the postfix notation of FORTH, was not designed with the ease of reading code in mind. As with FORTH, this prefix notation is relatively easy for a parser to decode - little more than lexical analysis is required - but ease of parsing is not the primary reason that LISP

source code is formatted the way it is. LISP code is written to reflect the fundamental structure of LISP data objects: linked lists which can be composed of atomic (indivisible) data objects of various types as well as of subordinate linked lists. User-defined LISP procedures are merely linked lists whose component parts are intelligible to an interpretation mechanism. As the linked list structure of LISP data objects forms access paths down to the level where actual data items, the atoms are located, so the linked list structure of LISP procedures forms access paths down to the level where executable code, the interpreter's primitive code sequences are found. The use of indirect pointers for the creation of access paths to executable code primitives makes LISP a variety of indirect-threaded code; the existence of multiple levels of indirection through dynamic data structures makes it a complex variety.

Since LISP is a far more popular language than FORTH I will assume that the reader has had some experience programming with it. Terminolgy which should be familiar to the LISP programmer will not be defined. The LISP source format and executable primitives are assumed to be as explained in the programming text by Winston and Horn.[43]

---

43. Winston and Horn, op. cit., (LISP).

This discussion will be centered almost entirely upon low-level LISP storage, access, and interpretation structures and algorithms. Furthermore, rather than treat an existing LISP implementation or a theoretical LISP internal structure scheme, I will cover the important points by explaining the organization and design decisions involved in the development of the LISP data manipulation function library which I wrote as part of the research/design work for this thesis. I will refer to this function library as LISPAK ("LISP package"). The package gives the C language programmer[44] on UNIX System V[45] the ability to manipulate LISP-style data structures using LISP-like notation and functions while allowing the package to handle run-time responsibilities such as pointer handling, error checking, storage allocation and garbage collection. The package does not support run-time interpretation of user-defined LISP secondaries; instead access to the list manipulating primitives is provided by way of function calls from compiled C code. This approach, which is an example of the use of subroutine threaded code to gain access to the LISP virtual machine, has the advantage of

---

44. See Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language (Englewood Cliffs, N.J.: Prentice Hall, 1978).

45. See Bell Telephone Laboratories, UNIX Programmer's Manual (New York, N.Y.: Holt, Rinehart and Winston, 1983).

providing machine code execution speed. The disadvantage of this approach is that the full expressive and information processing power of interpreted LISP is not supported, but I will outline the way in which a full LISP interpreter could be written in C using this powerful package as a base. Data manipulation functions currently included in the package are the following:[46] setq, boundp, null, atom, numberp, floatp, eq, equal, greatp, lessp, minusp, zerop, add1, sub1, plus, difference, times, quotient, remainder, minus, max, min, car, cdr, composites of car and cdr (caar through cddddr), cons, copy, rplaca, rplacd, nconc, append, delete, delq, explode, implode, assoc, last, length, member, reverse, subst, sort, sortcar, remob, putprop, get, remprop, quote, assorted housekeeping functions, several input and output functions and functions to provide conversions between C language data type formats and LISP internal data formats.

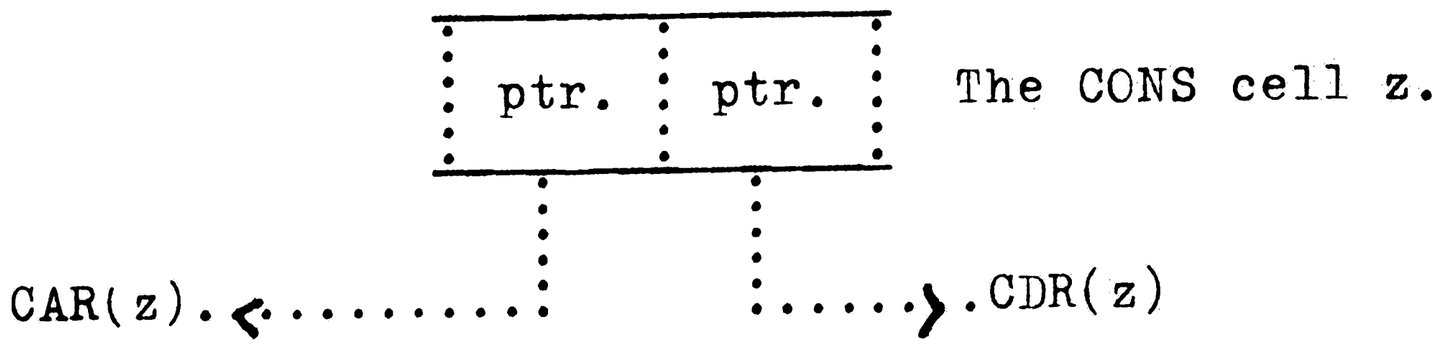
All of the internal structures of LISP are built from two fundamental types of data objects: atoms - these are the primary pieces of data out of which the lists of LISP are composed - and connecting cells, objects consisting of pointers used to connect atoms and other connecting cells. The connecting cells are called "cons cells" after

---

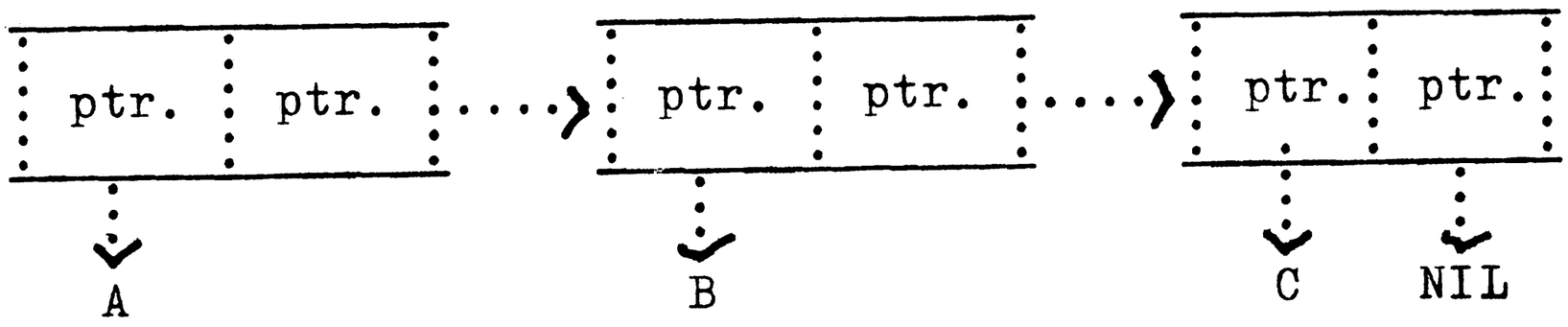
46. See Winston and Horn, op. cit., pages 409 through 412 for references to the actions of these primitive functions.

the constructor function "cons" which extends lists by allocating and interconnecting these cells.

> "The internal representation of LISP lists is built from primitives call CONS cells. Each CONS cell is an address that contains a pair of pointers, and each pointer can point either to an atom or to another CONS cell. In a typical LISP implementation, the CONS cells are computer words with pointers in their right and left halves, like the cell "z" diagrammed below:



> The left-half pointer points to the CAR of the cell z, the right half, to the CDR ... The list (A B C) is represented by three CONS cells whose left halves are used to link the cells together:



> In mathematics, sets are taken as the fundamental objects, and other concepts, such as ordered pairs, sequences, tuples, and relations, are defined in terms of sets. LISP data may be regarded as an alternative formalism in which the ordered pair, represented by the CONS cell, is fundamental. Sequences and sets are then represented by LISP lists, an n-tuple by a list of length n, and a relation by a list of tuples." [47]

---

47. Avron Barr and Edward A. Feigenbaum (eds.), "Programming Languages for AI Research: LISP," The Handbook of Artificial Intelligence, Volume 2 (Los Altos, CA.: William Kaufmann, Inc., 1982), p. 16-17.

From the LISP programmer's point of view the atom is the fundamental type of object since all LISP data values, whether simple atoms or lists, consist of atoms and the structure imposed on atoms by linkage. From the perspective of the LISP run-time system, however, the cons cell is the fundamental object since all access paths to atoms and executable primitives are by way of cons cell pointers.

In the LISPAK package the data type to which the package user has access, the "LISPVAR", is in fact a pointer to a cons cell. The LISPAK cons cell has the following format:

garbage collection marker - 1 bit position field

type of object indicated by car pointer - 2 bit field

car pointer - 29 bit field

cdr pointer - 32 bit field

The structure of the cons cell shown above takes advantage of machine-dependent characteristics of the computer on which the package was designed, the AT&T 3B20S minicomputer.[48] The 3B20S is a microcoded machine with thirty-two bit central processor registers and a hierarchy of semi-autonomous eight and sixteen bit input/output processors whose instruction set is optimized

---

48. See 3B20S/A Computers - IS25 Assembler Manual (Winston-Salem, N.C.: AT&T Technologies, 1984).

for the execution of compiled C programs. Storage is addressable in units of eight bit bytes, sixteen bit half-words, or thirty-two bit full-words. Pointers supported by the C compiler for this machine consume a full thirty-two bit word, but in fact the computer's hardware uses only the least significant twenty-four bits of a pointer when addressing memory; the top eight bits of a pointer are ignored by the hardware. LISPAK takes advantage of this fact by using one bit of the first word of the cons cell for garbage collection marking, two bits of the first word for run-time type checking, and the remaining twenty-nine bits for the car pointer; the car pointer is type cast (coerced) as an unsigned integer when stored and re-cast as a pointer when used; the re-casting fills the leftmost eight bits of the effective car pointer with zeroes. The cdr pointer consumes the full second thirty-two bit word. This organization will not work on a machine which uses the full thirty-two bits of a pointer for memory addressing. It is possible to allocate thirty-two bits for the car pointer in such machines and allow the remaining two fields to consume an additional three bit area. The problem with this approach is that alignment requirements on many computers would force alignment of cons cells such as these on full-word boundaries. Therefore if thirty-two bits were allocated for the car pointer and an additional three

for the other two fields, the compiler would allocate storage for the cell at a full-word boundary and insert twenty-nine padding bits between the first three bit positions and the car field, since the latter thirty-two bit pointer must be aligned on a full-word boundary. Placing the three bit position fields at the end of the cell would not solve the problem since full-word padding would then occur at the end of the cell. This problem brings us to the need for dynamic type checking and ways to implement it. Note that alignment is not a problem and storage is not consumed by padding on the 3B20S implementation of LISPAK because of the ability to pack a pointer and the three additional bit fields into a single full-word of storage.

The two-bit type tag of the LISPAK cons cell indicates to what type of object the car pointer points. Available types in LISPAK are literal atoms (stored as character strings), integer atoms, double precision floating point atoms, and other cons cells (the list composite type). The cdr pointer always points to another cons cell; the cell indicated by cdr is either the first cell in the rest of the list of which the current cell is an element, or a special cons cell representing NIL. Car can also point to NIL. When the car is used to retrieve data it is type cast to the appropriate kind of pointer; a great deal of the work of LISP is done by

comparing and manipulating pointers rather than linking to the atomic level for every operation.

One alternative to storing the type tag as part of the cons cell would be to store it with the data item at the destination of the car pointer. Unfortunately this would merely cause the same alignment problems for storage of the string atom, numeric atom or cons cell at the indicated address.

Another alternative would be to allocate arrays of cons cells consisting solely of car and cdr pointers of full-word size and allocate packed arrays of type tags to correspond with the cons cells. A type tag would be allocated for each cons cell, with type tags packed as many as possible to a machine word. The type of object indicated by the car of the third cons cell of a cons cell array would be found by unpacking and extracting the third type tag from the packed array. Extracting type tags from the packed array would consume processing time not required if the type tag is stored with the cons cell, but on machines using full-word pointers the separate type tag array approach would cut storage requirements for cons cell considerably.

Still another alternative is to set aside specific regions of memory for use as integer atoms, floating point atoms, literal atoms and cons cells. The type of the object can then be determined by testing the address

range of the pointer. Some inflexibility in terms of not being able to assign available memory to the wrong type of object is the major drawback of this method. Both MACLISP and INTERLISP use this method. The LISP machine gets around the problem by supporting a tagged architecture - type tags are stored as part of a cons cell which is defined to be the basic word of the machine; a full machine word includes a five-bit type tag field, so there is no alignment problem and wasted storage on the LISP machine. Run-time type checking is performed by microcode on this machine.[49]

The LISPAK package defines a user LISPVAR variable to be a pointer to a cons cell. Users of the package do not references or manipulate the internal individual fields of the cons cell directly, but supply the address or value of the LISPVAR pointer as required to the called primitive functions; data structure manipulation at the pointer level is invisible to the user.

A LISPVAR pointer is loaded with a value returned from a primitive constructor or selector function by calling the primitive "setq" with the address of the LISPVAR variable and the address of the cons cell to store there (the latter returned by function calls to other primitives in the package). Setq sets the LISPVAR

---

49. See Greenblatt, et. al., op. cit., p. 354.

pointer and stores the LISPVAR's address in an in-use list; the mark phase of the garbage collector uses the in-use list to determine which storage cells are referenced by active LISPVARs and which can be returned to the available cells free-list.

The LISPVAR pointer always points to a header cons cell. The car of this cons cell points to the value of the LISPVAR, which can be NIL, an integer atom, a double precision floating point atom, a literal atom string or another cons cell (a list value). The cdr of this header cons cell points to a property list. The property list is a chain of cons cells, terminated like all lists by a cdr pointer to the NIL cons cell. The property list alternates literal atoms representing property names (the car field of the list element points to the literal atom which is the property name, the cdr field to the rest of the property list) and elements representing property values (car points to the property value, cdr to the rest of the property list). Thus any LISPVAR inherently references two data structures, an implicit "value property" and the general property linked list.

Atomic values are stored uniquely in LISP; each distinct atom is stored only once. LISPAK presents no exception to this rule. Separate hash tables are maintained for literal atoms, integers, and floating point

atoms. New atoms are introduced into the system by calls to LISPAK input functions or by functions which convert C language strings, integers, floating point numbers and quoted lists into LISPVAR internal objects. When an atom is presented to LISPAK the package computes a hash number for the atom; the number represents an element in the hash array which is composed of pointers to distinct hash chains, one chain per hash bucket. Headers for atoms are stored in ascending sort order within a hash bucket chain. The atom input/conversion function scans the appropriate hash bucket chain until the insertion position for the atom is found. If a copy of the atom is already there then a pointer to the existing atom header is returned - the atom is not stored a second time - and this pointer will be loaded into the car field which references this atom. If the atom is new then storage for the atom header and the atomic value are dynamically allocated, the header is linked into the appropriate spot in the hash chain, the atomic value is linked or inserted into the header, and a pointer to the header is returned as in the case for an existing atom. Headers for atoms include a garbage collection marking bit, and headers which are no longer referenced by user LISPVAR variables will be marked and removed from the atomic hash lists during garbage collection.

LISPAK uses C pointer variables to point to linked

list data objects; the addresses of these pointer variables (but not their contents) are determined by the C compiler as static-global or stack-local references. Consequently LISPVAR variable names are not available at run-time, variable reference resolution having occurred at compile-time; the dynamic scoping rule for free variables in LISP, where a reference to a non-local variable refers to the most recently activated (and not yet deactivated) instance of that variable, does not apply to LISPAK.

Full LISP, on the other hand, requires access to variable names at run-time. The details of linkage of variable names to their values and properties may vary from implementation to implementation, but the concepts remain the same.

Examine Figure 4 on page 88. Atom headers for the literal atoms "A", "B", "car" and "userproc" are shown. The links coming in from the left hand side of the page might be coming from the hash bucket chains containing these atoms or from user variables which reference these atoms. Each LISP literal atom header in Figure 4 is linked to its local property list; as usual the property list is composed of alternating property names and values. If the literal atom is not used as a variable name then the property list will be empty; literal headers for atoms "value" and "code" on the right hand side of the

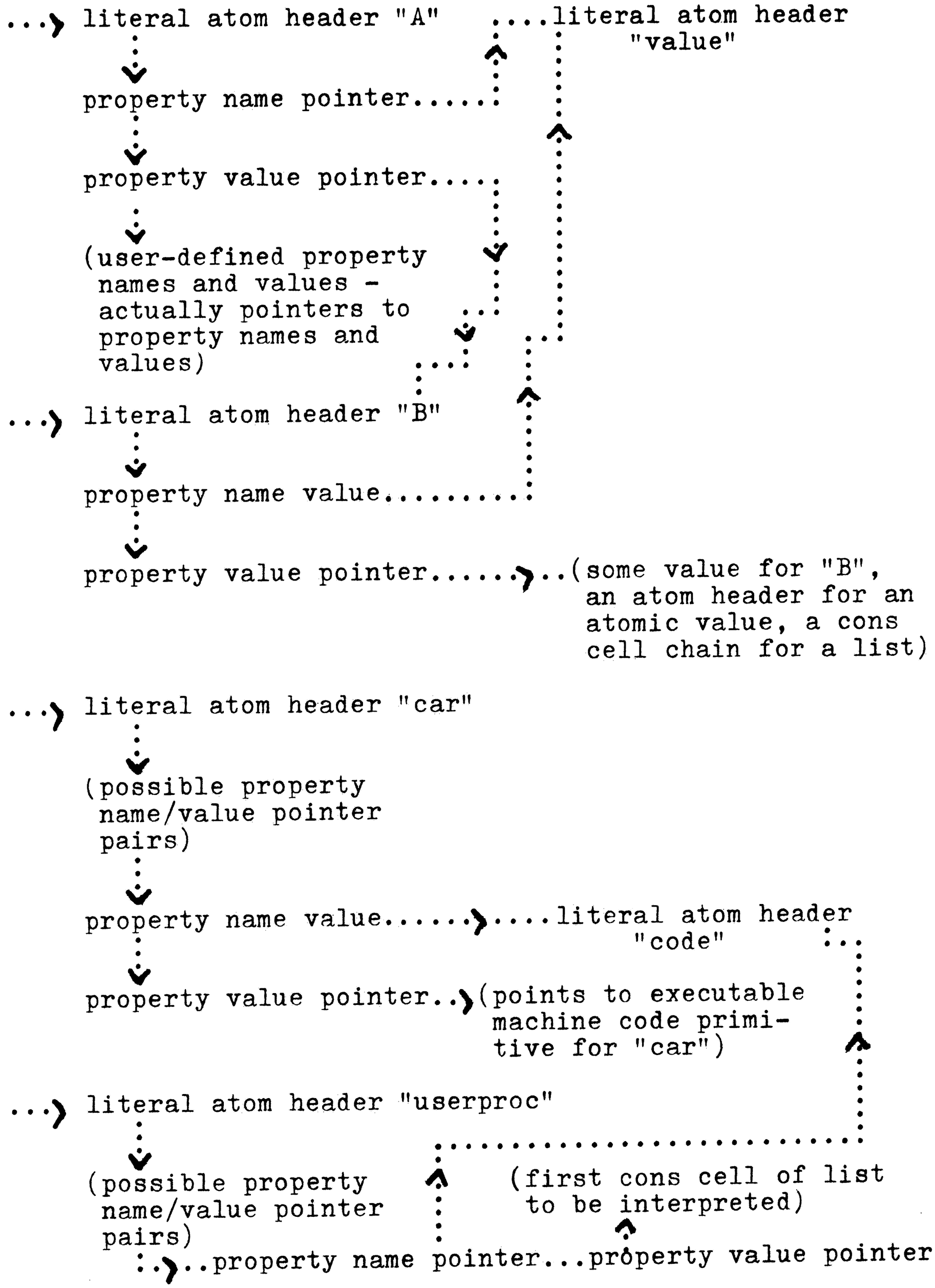


FIGURE 4 - LISP property list linkage

page are examples of literal atoms which are not used as variable identifiers. If the literal atom is used as a variable name then a property list will exist for that literal. Properties may include the default "value property" for the identifier as well as user defined property names and values.

A variable with a literal atom as its value has a value property pointer to that value literal header. In Figure 4 the value of variable "A" is the atomic value "B." Note that a property list chain extends from the header for "A", with a property name pointer pointing to the "value" header and a property value pointer pointing to the "B" header; other properties may also be defined for "A"; the property list can grow and shrink at run-time. While "B" serves as a value for "A", "B" serves as a variable in its own right; "B" also has a property name pointer to "value" and a corresponding value pointer going to some atom header or cons cell for a list value. The minimum requirement for a literal atom in LISP is that its string representation - called the "print name" of the atom - be linked to the literal atom header, and that the header be part of some larger access structure serving the same purpcse as the hash bucket chains in this example. Literal atoms which serve as variable names or property list names will be linked to a property list as well.

When a function call occurs some of the parameters and local variables of the newly called function may already have value properties bound in the calling function's environment. At function call time there must exist a method for saving the caller's environment's variable values on a stack if these same variable names are to be used locally by the called function without just over-writing the variable values of the caller's environment. Several methods of "deep binding" and "shallow binding" have been devised for saving a caller's variable values before binding variable names already in use to values local to a called function. Tradeoffs exist between the time required to perform a function call and the time required to reference a variable; time consuming function call mechanisms can be used to support faster variable access mechanisms. The details of binding strategies are beyond the scope of this essay and will not be examined in further detail here.[50]

Notice that in Figure 4 on page 88 the literal atoms "car" and "userproc" have among the properties on their property lists members which are called "code" properties. Hence we come to the structure by which LISP makes access

---

50. Interested readers should consult Henry G. Baker, Jr., "Shallow Binding in LISP 1.5," Artificial Intelligence: An MIT Perspective (Cambridge, MA.: MIT Press, 1979), p. 375-387.

to executable code possible: one (or possibly more) of the properties in a literal atom's property list can consist of a pointer to executable code. The indicated code will be a machine code sequence for the case of a primitive, or will be a list to be interpreted for the case of a secondary. Of course secondary lists may name procedures which also have secondaries for code property values, but as with all threaded code structures the interpreted secondary lists must eventually refer to machine primitives.

Since the "code" property of Figure 4 may refer to a secondary or primitive it is necessary to implement some method for distinguishing between primitive and secondary code property values, and for distinguishing between different possible varieties of primitives and secondaries. One way would be to expand the size of the type tag for the data item associated with the car pointer of the property value cons cell and create type tags for the different varieties of executable property values. Another scheme sometimes used is to assign a different property name for each type of possible executable code. Standard executable properties are: "SUBR" - a machine code primitive with a constant number of arguments whose arguments are evaluated before the function is called; "FSUBR" - a machine code primitive with a variable number of arguments whose arguments are not evaluated before

the function is called; "LSUBR" - a machine code primitive with a variable number of arguments whose arguments are evaluated before the function is called; "EXPR", "FEXPR", and "LEXPR" - the secondary, interpreted lists whose arguments' handling correspond respectively to SUBR, FSUBR, and LSUBR; and "MACRO" - a secondary which is used to modify a piece of code as data before having the code interpreted - compiler directives can be written by using the MACRO. The type tag method for distinguishing code types is less ambiguous than the distinct property name method since it is possible to define more than one executable property for a literal using the latter method.[51]

Thus it is possible that an identifier may refer to the value of a variable, the execution of a function, possibly both and more. How is it possible to know what use is intended when a literal identifier is used?

This brings us to the topic of interpretation, the subject which was the major focus during previous sections of this essay. It has been necessary to postpone the discussion of LISP interpretation until now because of the subordinate position which LISP code holds in relation to the overall structure of LISP data.

---

51. For more on storage structures, property lists and LISP code see John Allen, Anatomy of LISP (New York, N.Y.: McGraw-Hill, 1978), p. 244-361.

LISP is an interactive language, so its user interface centers around the activities of its command interpreter. LISP's command interpreter is essentially a "read-evaluate-print" loop. A symbolic expression (either an atom or a list) is read from the user's console or a file, the expression is interpreted by the inner interpreter, and the result is printed; all LISP functions return a result, and some have side-effects as well.

The heart of the interactive command interpreter is the inner interpreter, the "evaluate" function. When evaluate is given a numeric atom as an argument evaluate returns that numeric atom. When evaluate is given a literal atom then evaluate returns the value property of that literal, treating the literal as a variable identifier.

When evaluate is given a list then the situation becomes more complex. Evaluate considers a list to be a secondary function definition. Definitions can be stored in the code property slots of literal atoms using generic property creating functions such as "putprop", but normally special function-editing functions are written which aid the LISP programmer in writing a function list and storing it in the correct code property of its identifying literal atom. Note that when a function is being defined in LISP it is possible to make forward references to functions not yet defined; this was not so

with FORTH. When a reference to a function not yet defined is made in LISP the reference is only resolved to the extent that a pointer to the literal identifier for the incomplete function is installed in the secondary being defined. Of course it is necessary to complete the definition of the forward referenced function by filling in its code property before the referenced function can actually be used.

Given a list to interpret, evaluate considers the first data item in the list to be an executable function and the remaining data in the list to be the arguments of the function. If the first element of the list is a literal atom then evaluate retrieves its code property; normally the function name is required to be a literal atom, although conceivably if a list were supplied here the function evaluate could be applied recursively to the first element of the function list in order to resolve the literal name of the function.

Once the code property is retrieved evaluate determines whether it is the type of function which is to be called with evaluated arguments. If so then evaluate calls itself recursively to evaluate each of the arguments. Once the function's argument values have been resolved then evaluate calls the "apply" function to apply the requested function to the arguments by performing any modifications to the variable environment needed to

accomodate the arguments and then initiating execution of the called function; upon function termination control will be returned to apply, which can then return the environment to its previous state of binding (i.e. discard the argument values). The manner in which function execution is initiated depends upon the type of the function. A primitive function can simply be called with a machine code subroutine call. Secondaries are "called" by recursively calling evaluate after the arguments have been bound by apply. Evaluate must make provisions for the allocation of local variables and the interpretation of flow-of-control constructs such as "cond"-"go" iteration, "return" instructions and labels. Basic instructions within the called secondary consist of lists whose first element is the name of a primitive or secondary list, the latter which must be interpreted recursively. Eventually the interpretation process bottoms out in two ways: 1) primitive functions are referenced - these can be called directly without interpretation, and 2) instructions consisting of atoms rather than lists are reached, with numeric atoms yielding their values and literal atoms yielding their value property.

Extensions to this general interpretation scheme have been made in LISP implementations in order to allow user definition of primitive code segments and more explicit user control over the data environment in which

functions execute. The interested reader is urged to consult the information sources listed in the footnotes of this section.

Obviously the virtual LISP machine is, in terms of complexity of data structuring, a far cry from the virtual machine of FORTH. The two languages do exhibit some similarities, especially in those aspects which relate directly to interactive, immediate mode compilation and interpretation. One language emphasizes sparsity of code and speed of interpretation while the other emphasizes flexibility and power of data structuring. It is a tribute to the designers of both languages that they both have been implemented as microcode supported computing machines and that both languages have remained in use for many years after their creation.

CONCLUSIONS

With the increasing visibilty of even more powerful interpreted languages such as Smalltalk-80[52] it becomes obvious that the use of intermediate code schemes to support the development of new programming paradigms will undoubtedly continue to grow in popularity in the

---

52. Adele Goldberg and David Robson, Smalltalk-80, The Language and its Implementation (Reading, MA.: Addison Wesley, 1983).

future. Data driven languages and actor and object oriented languages can all make use of facilities readily provided by intermediate code structures. No longer is it the case that the poor programming language and compiler designers must suffer with computer architectures which cater more to the requirements of primitive electronic circuits than to the needs of a run-time software environment. More and more,computer scientists and software engineers are contributing to the design of computer architectures, but before an architecture can be committed to hardware it must be tested, revised, extended, and tested some more. Intermediate codes provide a way in which novel computer architectures can be implemented first as virtual computers, a simpler and less expensive route than the immediate design of a new hardware computer upon conception of a computer architecture. LISP and FORTH were virtual computers a long time before they were first supported by microcode. The languages proved their worth, and in doing so proved the worth of the threaded code intermediate virtual computer as a tool for use in designing both programming languages and computer architectures.

BIBLIOGRAPHY

Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Reading, MA.: Addison Wesley, 1977.

Allen, John. Anatomy of LISP. New York, NY.: McGraw-Hill, 1978.

Baker, Henry G., Jr. "Shallow Binding in LISP 1.5", Artificial Intelligence: An MIT Perspective, Volume 2. Patrick H. Winston and Richard H. Brown, editors. Cambridge, MA.: MIT Press, 1979, p. 377-387.

Barr, Avron and Edward A. Feigenbaum, editors. "Programming Languages for AI Research: LISP", The Handbook of Artificial Intelligence, Volume 2. Los Altos, CA.: William Kaufmann, Inc., 1982, p. 15-29.

Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Chicago, IL.: Science Research Associates, 1979.

Bawden, Alan, Richard Greenblatt, John Holloway, Tom Knight, David Moon and Daniel Weinreb. "The LISP Machine", Artificial Intelligence: An MIT Perspective, Volume 2. Patrick H. Winston and Richard H. Brown, editors. Cambridge, MA.: MIT Press, 1979, p. 347-373.

Bell Laboratories. 3B20S/A Computers - IS25 Assembler Manual. Winston-Salem, NC.: AT&T Technologies, 1984.

Bell Laboratories. UNIX Programmer's Manual. New York, NY.: Holt, Rinehart and Winston, 1983.

Brown, P. J. Writing Interactive Compilers and Interpreters. New York, NY.: John Wiley and Sons, 1979.

Eckhouse, Richard H., Jr. and L. Robert Morris. Minicomputer Systems - Organization, Programming, and Applications (PDP-11). Englewood Cliffs, NJ.: Prentice Hall, 1979.

Goldberg, Adele and David Robson. Smalltalk-80, The Language and its Implementation. Reading, MA.: Addison Wesley, 1983.

Harris, Kim. "FORTH Extensibility", BYTE, Volume 5, Number 8 (August, 1980), p. 164-184.

Highberger, Deb and Dan Edson. "Intelligent Computing Era Takes Off", Computer Design, Volume 23, Number 10 (September, 1984), p. 79-95.

Hindin, Harvey J. "Revolution Brewing in Workstation Technology", Computer Design, Volume 24, Number 1 (January, 1985), p. 111-124.

Hindin, Harvey J. "Fifth Generation Computing: Dedicated Software is the Key", Computer Design, Volume 23, Number 10 (September, 1984), p. 150-164.

James, John S. FIG-FORTH for PDP-11 Assembly Source Listing. San Carlos, CA.: FORTH Interest Group, 1979.

Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs, NJ.: Prentice Hall, 1978.

Kraft, George D. and Wing N. Toy. Microprogrammed Control and Reliable Design of Small Computers. Englewood Cliffs, NJ.: Prentice Hall, 1981.

Loeliger, R.G. Threaded Interpretive Languages. Peterborough, NH.: BYTE Publications, 1981.

Mokhoff, Nicolas. "Parallelism Makes Strong Bid for Next Generation Computers", Computer Design, Volume 23, Number 10 (September, 1984), p. 104-131.

Moore, Charles H. "The Evolution of FORTH, An Unusual Language", BYTE, Volume 5, Number 8 (August, 1980), p. 76-90.

Pratt, Terrence W. Programming Languages: Design and Implementation. Englewood Cliffs, NJ.: Prentice Hall, 1984.

Ragsdale, William F. FIG-FORTH 6502 Assembly Source Listing. San Carlos, CA.: FORTH Interest Group, 1980.

Ragsdale, William F. FIG-FORTH Installation Manual. San Carlos, CA.: FORTH Interest Group, 1980.

Rather, E., L. Brodie and C. Rosenberg. Using FORTH. Hermosa Beach, CA.: FORTH, Inc., 1980.

Rector, Russell and George Alexy. The 8086 Book. Berkeley, CA.: Osborne/McGraw-Hill, 1980.

Ritter, Terry and Gregory Walker. "Varieties of Threaded Code for Language Implementation", BYTE, Volume 5, Number 9 (September, 1980), p. 206-227.

Tanenbaum, Andrew S. Structured Computer Organization. Englewood Cliffs, NJ.: Prentice Hall, 1976.

Ting, C. H. Systems Guide to FIG-FORTH. San Mateo, CA.: Offete Enterprises, 1980.

Winston, Patrick Henry and Berthold Klaus Paul Horn. LISP. Reading, MA.: Addison Wesley, 1981.

BIOGRAPHICAL INFORMATION

My name is Dale Edward Parson, born to Ada Rose (Kline) Parson and Robert Roscoe Parson, Sr. in Reading, Pennsylvania on November 24, 1954. On August 9, 1975 Linda Carol Millard of Oley, Pennsylvania and I were wed. We are both vegetarian Buddhists who are committed to the nonviolent study and application of intelligent awareness. My interest in artificial intelligence derives from this orientation.

I received my high school diploma from Oley Valley High School, Oley, Pennsylvania in 1972. My Bachelor of Science degree was awarded by Albright College, Reading, Pennsylvania in 1983. Over one half of my undergraduate credits and all of my graduate credits obtained so far were earned on a part-time basis while I have been employed full-time. In 1983 I was inducted into the Alpha Sigma Lambda honor society for students of continuing education as a charter member of the Beta Phi Chapter at Albright. I am also a member of American Mensa.

I have been employed as an electronics technician at AT&T Technologies in Reading, Pennsylvania since 1979. While interested in industrial applications of computers, I am equally interested in the possibility of teaching computer science some day.